
# PCI Local Bus Specification

**Revision 2.2**

**December 18, 1998**

| REVISION | REVISION HISTORY | DATE |
|---|---|---|
| 1.0 | Original issue | 6/22/92 |
| 2.0 | Incorporated connector and expansion board specification | 4/30/93 |
| 2.1 | Incorporated clarifications and added 66 MHz chapter | 6/1/95 |
| 2.2 | Incorporated ECNs and improved readability | 12/18/98 |

The PCI Special Interest Group disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does the PCI Special Interest Group make a commitment to update the information contained herein.

Contact the PCI Special Interest Group office to obtain the latest revision of the specification.

Questions regarding the PCI specification or membership in the PCI Special Interest Group may be forwarded to:

PCI Special Interest Group
2575 N.E. Kathryn #17
Hillsboro, Oregon 97124
Phone: 800-433-5177 (Inside the U.S.)
503-693-6360 (Outside the U.S.)
Fax: 503-693-8344
e-mail pcisig@pcisig.com
http://www.pcisig.com

## DISCLAIMER

# Contents

## Chapter 1 Introduction



## Chapter 2 Signal Definition

# Chapter 3 Bus Operation

# Chapter 4 Electrical Specification

## Chapter 5 Mechanical Specification



## Chapter 6 Configuration Space

## Chapter 7 66 Mhz PCI Specification

# Figures

# Preface

## Specification Supersedes Earlier Documents

This document contains the formal specifications of the protocol, electrical, and mechanical features of the *PCI Local Bus Specification, Revision 2.2*, as the production version effective December 18, 1998. The *PCI Local Bus Specification, Revision 2.1,* issued June 1, 1995, is superseded by this specification.

Following publication of the *PCI Local Bus Specification, Revision 2.2*, there may be future approved errata and/or approved changes to the specification prior to the issuance of another formal revision. To assure designs meet the latest level requirements, designers of PCI devices must refer to the PCI SIG home page at http://www.pcisig.com, in the members-only section, for any approved changes.

## Incorporation of Engineering Change Requests (ECRs)

The following ECRs have been incorporated into this production version of the specification:

| ECR | Description |
|---|---|
| New Capabilities | Changes to configuration structure to define additional capabilities of a PCI function |
| Sub ID | Subsystem vendor configuration space changed from optional to required for most classes of devices |
| **PME#** | Describes wake-up function and assigns (previously reserved) pin on PCI bus |
| MECH ECN# 1 | Bracket mounting for EMI reduction |
| MECH ECN# 2 | Increase size of I/O window to enable new connectors |
| MECH ECN# 3 | I/O connector volume to enable add-in card insertion |
| MECH ECN# 4 | Riser connector-add-in card interface |

| ECR | Description |
|---|---|
| MECH ECN# 5 | Tighten critical tolerance to improve add-in card seating |
| MECH ECN# 6 | Mechanical errata (wrong datum ref), components free areas |
| Posted Memory Writes | Clarified requirements for posted memory writes to prevent situations which may lead to a deadlock |
| Tprop | Clarifies measurement and determination of Tprop times for 33/66 MHz |
| **RST#** TIMING | New timing requirements between **RST#** and the first transaction on the bus and the first access to a device |
| VPD (Revised) | Provides alternate access method to vital product data |
| 3.3 V Aux | Specifies a standard source of power for power management wake event logic |
| Max Retry Time | Devices cannot Retry a memory write request for longer than 10 µs |
| Msg Interrupt | Method by which an I/O controller can deliver message based interrupt |
| Spread Spectrum Clocking | Adds spread spectrum clocking (SSC) to the specification |

## Document Conventions

The following name and usage conventions are used in this document:

| | |
|---|---|
| asserted, deasserted | The terms *asserted* and *deasserted* refer to the globally visible state of the signal on the clock edge, not to signal transitions. |
| edge, clock edge | The terms *edge* and *clock edge* refer to the rising edge of the clock. On the rising edge of the clock is the only time signals have any significance on the PCI bus. |
| # | A # symbol at the end of a signal name indicates that the signal's asserted state occurs when it is at a low voltage. The absence of a # symbol indicates that the signal is asserted at a high voltage. |
| reserved | The contents or undefined states or information are not defined at this time. Using any reserved area in the PCI specification is not permitted. All areas of the PCI specification can only be changed according to the by-laws of the PCI Special Interest Group. Any use of the reserved areas of the PCI specification will result in a product that is not PCI-compliant. The functionality of any such product cannot be guaranteed in this or any future revision of the PCI specification. |
| signal names | Signal names are indicated with **this bold** font. |
| signal range | A signal name followed by a range enclosed in brackets, for example **AD[31::00]**, represents a range of logically related signals. The first number in the range indicates the most significant bit (msb) and the last number indicates the least significant bit (lsb). |
| implementation notes | Implementation notes are enclosed in a box. They are not part of the PCI specification and are included for clarification and illustration only. |


# Chapter 3
# Bus Operation

## 3.1. Bus Commands

Bus commands indicate to the target the type of transaction the master is requesting. Bus commands are encoded on the **C/BE[3::0]#** lines during the address phase.

### 3.1.1. Command Definition

PCI bus command encodings and types are listed below, followed by a brief description of each. Note: The command encodings are as viewed on the bus where a "1" indicates a high voltage and "0" is a low voltage. Byte enables are asserted when "0".

| C/BE[3::0]# | Command Type |
|---|---|
| 0000 | Interrupt Acknowledge |
| 0001 | Special Cycle |
| 0010 | I/O Read |
| 0011 | I/O Write |
| 0100 | Reserved |
| 0101 | Reserved |
| 0110 | Memory Read |
| 0111 | Memory Write |
| 1000 | Reserved |
| 1001 | Reserved |
| 1010 | Configuration Read |
| 1011 | Configuration Write |
| 1100 | Memory Read Multiple |
| 1101 | Dual Address Cycle |
| 1110 | Memory Read Line |
| 1111 | Memory Write and Invalidate |

The *Interrupt Acknowledge* command is a read implicitly addressed to the system interrupt controller. The address bits are logical don't cares during the address phase and the byte enables indicate the size of the vector to be returned.

The *Special Cycle* command provides a simple message broadcast mechanism on PCI. It is designed to be used as an alternative to physical signals when sideband communication is necessary. This mechanism is fully described in Section 3.6.2.

The *I/O Read* command is used to read data from an agent mapped in I/O Address Space. **AD[31::00]** provide a byte address. All 32 bits must be decoded. The byte enables indicate the size of the transfer and must be consistent with the byte address.

The *I/O Write* command is used to write data to an agent mapped in I/O Address Space. All 32 bits must be decoded. The byte enables indicate the size of the transfer and must be consistent with the byte address.

*Reserved* command encodings are reserved for future use. PCI targets must not alias reserved commands with other commands. Targets must not respond to reserved encodings. If a reserved encoding is used on the interface, the access typically will be terminated with Master-Abort.

The *Memory Read* command is used to read data from an agent mapped in the Memory Address Space. The target is free to do an anticipatory read for this command only if it can guarantee that such a read will have no side effects. Furthermore, the target must ensure the coherency (which includes ordering) of any data retained in temporary buffers after this PCI transaction is completed. Such buffers must be invalidated before any synchronization events (e.g., updating an I/O status register or memory flag) are passed through this access path.

The *Memory Write* command is used to write data to an agent mapped in the Memory Address Space. When the target returns "ready," it has assumed responsibility for the coherency (which includes ordering) of the subject data. This can be done either by implementing this command in a fully synchronous manner, or by insuring any software transparent posting buffer will be flushed before synchronization events (e.g., updating an I/O status register or memory flag) are passed through this access path. This implies that the master is free to create a synchronization event immediately after using this command.

The *Configuration Read* command is used to read the Configuration Space of each agent. An agent is selected during a configuration access when its **IDSEL** signal is asserted and **AD[1::0]** are 00. During the address phase of a configuration transaction, **AD[7::2]** address one of the 64 DWORD registers (where byte enables address the byte(s) within each DWORD) in Configuration Space of each device and **AD[31::11]** are logical don't cares to the selected agent (refer to Section 3.2.2.3.). **AD[10::08]** indicate which device of a multi-function agent is being addressed.

The *Configuration Write* command is used to transfer data to the Configuration Space of each agent. Addressing for configuration write transactions is the same as for configuration read transactions.

The *Memory Read Multiple* command is semantically identical to the Memory Read command except that it additionally indicates that the master may intend to fetch more than one cacheline before disconnecting. The memory controller continues pipelining memory requests as long as **FRAME#** is asserted. This command is intended to be used with bulk sequential data transfers where the memory system (and the requesting master) might gain some performance advantage by sequentially reading ahead one or more additional cacheline(s) when a software transparent buffer is available for temporary storage.

The *Dual Address Cycle (DAC)* command is used to transfer a 64-bit address to devices that support 64-bit addressing when the address is not in the low 4-GB address space.

Targets that support only 32-bit addresses must treat this command as reserved and not respond to the current transaction in any way.

The *Memory Read Line* command is semantically identical to the Memory Read command except that it additionally indicates that the master intends to fetch a complete cacheline. This command is intended to be used with bulk sequential data transfers where the memory system (and the requesting master) might gain some performance advantage by reading up to a cacheline boundary in response to the request rather than a single memory cycle. As with the Memory Read command, pre-fetched buffers must be invalidated before any synchronization events are passed through this access path.

The *Memory Write and Invalidate* command is semantically identical to the Memory Write command except that it additionally guarantees a minimum transfer of one complete cacheline; i.e., the master intends to write all bytes within the addressed cacheline in a single PCI transaction unless interrupted by the target. Note: All byte enables must be asserted during each data phase for this command. The master may allow the transaction to cross a cacheline boundary only if it intends to transfer the entire next line also. This command requires implementation of a configuration register in the master indicating the cacheline size (refer to Section 6.2.4. for more information) and may only be used with Linear Burst Ordering (refer to Section 3.2.2.2.). It allows a memory performance optimization by invalidating a "dirty" line in a write-back cache without requiring the actual write-back cycle thus shortening access time.

## 3.1.2. Command Usage Rules

All PCI devices (except host bus bridges) are required to respond as a target to configuration (read and write) commands. All other commands are optional.

A master may implement the optional commands as needed. A target may also implement the optional commands as needed, but if it implements basic memory commands, it must support all the memory commands, including Memory Write and Invalidate, Memory Read Line, and Memory Read Multiple. If not fully implemented, these performance optimizing commands must be aliased to the basic memory commands. For example, a target may not implement the Memory Read Line command; however, it must accept the request (if the address is decoded for a memory access) and treat it as a Memory Read command. Similarly, a target may not implement the Memory Write and Invalidate command, but must accept the request (if the address is decoded for a memory access) and treat it as a Memory Write command.

For block data transfers to/from system memory, Memory Write and Invalidate, Memory Read Line, and Memory Read Multiple are the recommended commands for masters capable of supporting them. The Memory Read or Memory Write commands can be used if for some reason the master is not capable of using the performance optimizing commands. For masters using the memory read commands, any length access will work for all commands; however, the preferred use is shown below.

While Memory Write and Invalidate is the only command that requires implementation of the Cacheline Size register, it is strongly suggested the memory read commands use it as well. A bridge that prefetches is responsible for any latent data not consumed by the master.

Memory command recommendations vary depending on the characteristics of the memory location and the amount of data being read. Memory locations are characterized as either *prefetchable* or *non-prefetchable*. Prefetchable memory has the following characteristics:

- There are no side effects of a read operation. The read operation cannot be destructive to either the data or any other state information. For example, a FIFO that advances to the next data when read would not be prefetchable. Similarly, a location that cleared a status bit when read would not be prefetchable.
- When read, the device is required to return all bytes regardless of the byte enables (four or eight depending upon the width of the data transfer (refer to Section 3.8.1.)).
- Bridges are permitted to merge writes into this range (refer to Section 3.2.3.).

All other memory is considered to be non-prefetchable.

The preferred use of the read commands is:

| | |
|---|---|
| Memory Read | When reading data in an address range that has side-effects (not prefetchable) or reading a single DWORD |
| Memory Read Line | Reading more than a DWORD up to the next cacheline boundary in a prefetchable address space |
| Memory Read Multiple | Reading a block which crosses a cacheline boundary (stay one cacheline ahead of the master if possible) of data in a prefetchable address range |

The target should treat the read commands the same even though they do not address the first DWORD of the cacheline. For example, a target that is addressed at DWORD 1 (instead of DWORD 0) should only prefetch to the end of the current cacheline. If the Cacheline Size register is not implemented, then the master should assume a cacheline size of either 16 or 32 bytes and use the read commands recommended above. (This assumes linear burst ordering.)

### Implementation Note: Using Read Commands

Different read commands will have different affects on system performance because host bridges and PCI-to-PCI bridges must treat the commands differently. When the Memory Read command is used, a bridge will generally obtain only the data the master requested and no more since a side-effect may exist. The bridge cannot read more because it does not know which bytes are required for the next data phase. That information is not available until the current data phase completes. However, for Memory Read Line and Memory Read Multiple, the master guarantees that the address range is prefetchable, and, therefore, the bridge can obtain more data than the master actually requested. This process increases system performance when the bridge can prefetch and the master requires more than a single DWORD. (Refer to the PCI-PCI Bridge Architecture Specification for additional details and special cases.)

As an example, suppose a master needed to read three DWORDs from a target on the other side of a PCI-to-PCI bridge. If the master used the Memory Read command, the bridge could not begin reading the second DWORD from the target because it does not have the next set of byte enables and, therefore, will terminate the transaction after a single data transfer. If, however, the master used the Memory Read Line command, the bridge would be free to burst data from the target through the end of the cacheline allowing the data to flow to the master more quickly.

The Memory Read Multiple command allows bridges to prefetch data farther ahead of the master, thereby increasing the chances that a burst transfer can be sustained.

It is highly recommended that the Cacheline Size register be implemented to ensure correct use of the read commands. The Cacheline Size register must be implemented when using the optional Cacheline Wrap mode burst ordering.

Using the correct read command gives optimal performance. If, however, not all read commands are implemented, then choose the ones which work the best most of the time. For example, if the large majority of accesses by the master read entire cachelines and only a small number of accesses read more than a cacheline, it would be reasonable for the device to only use the Memory Read Line command for both types of accesses.

A bridge that prefetches is responsible for any latent data not consumed by the master. The simplest way for the bridge to correctly handle latent data is to simply mark it invalid at the end of the current transaction.

### Implementation Note: Stale-Data Problems Caused By Not Discarding Prefetch Data

Suppose a CPU has two buffers in adjacent main memory locations. The CPU prepares a message for a bus master in the first buffer and then signals the bus master to pick up the message. When the bus master reads its message, a bridge between the bus master and main memory prefetches subsequent addresses, including the second buffer location.

Some time later the CPU prepares a second message using the second buffer in main memory and signals the bus master to come and get it. If the intervening bridge has not flushed the balance of the previous prefetch, then when the master attempts to read the second buffer the bridge may deliver stale data.

Similarly, if a device were to poll a memory location behind a bridge, the device would never observe a new value of the location if the bridge did not flush the buffer after each time the device read it.

## 3.2. PCI Protocol Fundamentals

The basic bus transfer mechanism on PCI is a burst. A burst is composed of an address phase and one or more data phases. PCI supports bursts in both Memory and I/O Address Spaces.

All signals are sampled on the rising edge of the clock[9]. Each signal has a setup and hold aperture with respect to the rising clock edge, in which transitions are not allowed. Outside this aperture, signal values or transitions have no significance. This aperture occurs only on "qualified" rising clock edges for **AD[31::00]**, **AD[63::32]**, **PAR**[10], **PAR64**, and **IDSEL** signals[11] and on every rising clock edge for **LOCK#**, **IRDY#**, **TRDY#**, **FRAME#**, **DEVSEL#**, **STOP#**, **REQ#**, **GNT#**, **REQ64#**, **ACK64#**, **SERR#** (on the falling edge of **SERR#** only), and **PERR#**. **C/BE[3::0]#**, **C/BE[7::4]#** (as bus commands) are qualified on the clock edge that **FRAME#** is first asserted. **C/BE[3::0]#**, **C/BE[7::4]#** (as byte enables) are qualified on each rising clock edge following the completion of an address phase or data phase and remain valid the entire data phase. **RST#**, **INTA#**, **INTB#**, **INTC#**, and **INTD#** are not qualified nor synchronous.

### 3.2.1. Basic Transfer Control

The fundamentals of all PCI data transfers are controlled with three signals (see Figure 3-5).

**FRAME#** is driven by the master to indicate the beginning and end of a transaction.

**IRDY#** is driven by the master to indicate that it is ready to transfer data.

**TRDY#** is driven by the target to indicate that it is ready to transfer data.

The interface is in the Idle state when both **FRAME#** and **IRDY#** are deasserted. The first clock edge on which **FRAME#** is asserted is the *address phase*, and the address and bus command code are transferred on that clock edge. The next[12] clock edge begins the first of one or more *data phases* during which data is transferred between master and target on each clock edge for which both **IRDY#** and **TRDY#** are asserted. Wait cycles may be inserted in a data phase by either the master or the target when **IRDY#** or **TRDY#** is deasserted.

The source of the data is required to assert its ***x*RDY#** signal unconditionally when data is valid (**IRDY#** on a write transaction, **TRDY#** on a read transaction). The receiving agent may delay the assertion of its ***x*RDY#** when it is not ready to accept data. When delaying the assertion of its ***x*RDY#**, the target and master must meet the latency requirements specified in Sections 3.5.1.1. and 3.5.2. In all cases, data is only transferred when **IRDY#** and **TRDY#** are both asserted on the same rising clock edge.

---

[9] The only exceptions are **RST#**, **INTA#**, **INTB#**, **INTC#**, and **INTD#** which are discussed in Sections 2.2.1. and 2.2.6.

[10] **PAR** and **PAR64** are treated like an **AD** line delayed by one clock.

[11] The notion of qualifying **AD** and **IDSEL** signals is fully defined in Section 3.6.3.

[12] Note: The address phase consists of two clocks when the command is the Dual Address Cycle (DAC).

Once a master has asserted **IRDY#**, it cannot change **IRDY#** or **FRAME#** until the current data phase completes regardless of the state of **TRDY#**. Once a target has asserted **TRDY#** or **STOP#**, it cannot change **DEVSEL#**, **TRDY#**, or **STOP#** until the current data phase completes. Neither the master nor the target can change its mind once it has committed to the current data transfer until the current data phase completes. (A data phase completes when **IRDY#** and [**TRDY#** or **STOP#**] are asserted.) Data may or may not transfer depending on the state of **TRDY#**.

At such time as the master intends to complete only one more data transfer (which could be immediately after the address phase), **FRAME#** is deasserted and **IRDY#** is asserted indicating the master is ready. After the target indicates that it is ready to complete the final data transfer (**TRDY#** is asserted), the interface returns to the Idle state with both **FRAME#** and **IRDY#** deasserted.

## 3.2.2. Addressing

PCI defines three physical address spaces. The *Memory* and *I/O Address Spaces* are customary. The *Configuration Address Space* has been defined to support PCI hardware configuration. Accesses to this space are further described in Section 3.2.2.3.

PCI targets (except host bus bridges) are required to implement Base Address register(s) to request a range of addresses which can be used to provide access to internal registers or functions (refer to Chapter 6 for more details). The configuration software uses the Base Address register to determine how much space a device requires in a given address space and then assigns (if possible) where in that space the device will reside.

### Implementation Note: Device Address Space

It is highly recommended, that a device request (via Base Address register(s)) that its internal registers be mapped into Memory Space and not I/O Space. Although the use of I/O Space is allowed, I/O Space is limited and highly fragmented in PC systems and will become more difficult to allocate in the future. Requesting Memory Space instead of I/O Space allows a device to be used in a system that does not support I/O Space. A device may map its internal register into both Memory Space and optionally I/O Space by using two Base Address registers, one for I/O and the other for Memory. The system configuration software will allocate (if possible) space to each Base Address register. When the agent's device driver is called, it determines which address space is to be used to access the device. If the preferred access mechanism is I/O Space and the I/O Base Address register was initialized, then the driver would access the device using I/O bus transactions to the I/O Address Space assigned. Otherwise, the device driver would be required to use memory accesses to the address space defined by the Memory Base Address register. Note: Both Base Address registers provide access to the same registers internally.

When a transaction is initiated on the interface, each potential target compares the address with its Base Address register(s) to determine if it is the target of the current transaction. If it is the target, the device asserts **DEVSEL#** to claim the access. For more details about **DEVSEL#** generation, refer to Section 3.6.1. How a target completes address decode in each address space is discussed in the following sections.

### 3.2.2.1. I/O Space Decoding

In the I/O Address Space, all 32 **AD** lines are used to provide a full byte address. The master that initiates an I/O transaction is required to ensure that **AD[1::0]** indicate the least significant valid byte for the transaction.

The byte enables indicate the size of the transfer and the affected bytes within the DWORD and must be consistent with **AD[1::0]**. Table 3-1 illustrates the valid combinations for **AD[1::0]** and the byte enables for the initial data phase.

**Table 3-1: Byte Enables and AD[1::0] Encodings**

| **AD[1::0]** | Starting Byte | Valid **BE#[3:0]** Combinations |
|---|---|---|
| 00 | Byte 0 | xxx0 or 1111 |
| 01 | Byte 1 | xx01 or 1111 |
| 10 | Byte 2 | x011 or 1111 |
| 11 | Byte 3 | 0111 or 1111 |

Note if **BE#[3::0]** = 1111, **AD[1::0]** can have any value.

A function may restrict what type of access(es) it supports in I/O Space. For example, a device may restrict its driver to only access the function using byte, word, or DWORD operations and is free to terminate all other accesses with Target-Abort. How a device uses **AD[1::0]** and **BE#[3::0]** to determine which accesses violate its addressing restrictions is implementation specific.

A device (other than an expansion bus bridge) that claims legacy I/O addresses whenever its I/O Space enable bit is set (i.e., without the use of Base Address Registers) is referred to as a legacy I/O device. Legacy I/O devices are discussed in Appendix G.

### 3.2.2.2. Memory Space Decoding

In the Memory Address Space, the **AD[31::02]** bus provides a DWORD aligned address. **AD[1::0]** are not part of the address decode. However, **AD[1::0]** indicate the order in which the master is requesting the data to be transferred.

Table 3-2 lists the burst ordering requested by the master during Memory commands as indicated on **AD[1::0]**.

**Table 3-2: Burst Ordering Encoding**

| AD1 | AD0 | Burst Order |
|---|---|---|
| 0 | 0 | Linear Incrementing |
| 0 | 1 | Reserved (disconnect after first data phase)[13] |
| 1 | 0 | Cacheline Wrap mode |
| 1 | 1 | Reserved (disconnect after first data phase) |

All targets are required to check **AD[1::0]** during a memory command transaction and either provide the requested burst order or terminate the transaction with Disconnect in one of two ways. The target can use Disconnect With Data during the initial data phase or Disconnect Without Data for the second data phase. With either termination, only a single data phase transfers data. The target is not allowed to terminate the transaction with Retry solely because it does not support a specific burst order. If the target does not support the burst order requested by the master, the target must complete one data phase and then terminate the request with Disconnect. This ensures that the transaction will complete (albeit slowly, since each request will complete as a single data phase transaction). If a target supports bursting on the bus, the target must support the linear burst ordering. Support for cacheline wrap is optional.

In linear burst order mode, the address is assumed to increment by one DWORD (four bytes) for 32-bit transactions and two DWORDs (eight bytes) for 64-bit transactions after each data phase until the transaction is terminated (an exception is described in Section 3.9.). Transactions using the Memory Write and Invalidate command can only use the linear incrementing burst mode.

A cacheline wrap burst may begin at any address offset within the cacheline. The length of a cacheline is defined by the Cacheline Size register (refer to Section 6.2.4.) in Configuration Space which is initialized by configuration software. The access proceeds by incrementing one DWORD address (two DWORDS for a 64-bit data transaction) until the end of the cacheline has been reached, and then wraps to the beginning of the same cacheline. It continues until the rest of the line has been transferred. For example, an access where the cacheline size is 16 bytes (four DWORDs) and the transaction addresses DWORD 08h, the sequence for a 32-bit transaction would be:

First data phase is to DWORD 08h

Second data phase is to DWORD 0Ch (which is the end of the current cacheline)

Third data phase is to DWORD 00h (which is the beginning of the addressed cacheline)

Last data phase is to DWORD 04h (which completes access to the entire cacheline)

[13] This encoded value is reserved and cannot be assigned any "new" meaning for new designs. New designs (master or targets) cannot use this encoding. Note that in an earlier version of this specification, this encoding had meaning and there are masters that generate it and some targets may allow the transaction to continue past the initial data phase.

If the burst continues once a complete cacheline has been accessed, the burst continues at the same DWORD offset of the next cacheline. Continuing the burst of the previous example would be:

Fifth data phase is to DWORD 18h

Sixth data phase is to DWORD 1Ch (which is the end of the second cacheline)

Seventh data phase is to DWORD 10h (which is the beginning of the second cacheline)

Last data phase is to DWORD 14h (which completes access to the second cacheline)

If a target does not implement the Cacheline Size register, the target must signal Disconnect with or after the completion of the first data phase if Cacheline Wrap or a reserved mode is used.

If a master starts with one burst ordering, it cannot change the burst ordering until the current transaction ends, since the burst ordering information is provided on **AD[1::0]** during the address phase.

A device may restrict what access granularity it supports in Memory Space. For example, a device may restrict its driver to only access the device using byte, word, or DWORD operations and is free to terminate all other accesses with Target-Abort.

### 3.2.2.3. Configuration Space Decoding

Every device, other than host bus bridges, must implement Configuration Address Space. Host bus bridges may optionally implement Configuration Address Space. In the Configuration Address Space, each function is assigned a unique 256-byte space that is accessed differently than I/O or Memory Address Spaces. Configuration registers are described in Chapter 6. The following sections describe:

- Configuration commands (Type 0 and Type 1)
- Software generation of configuration commands
- Software generation of Special Cycles
- Selection of a device's Configuration Space
- System generation of **IDSEL**

#### 3.2.2.3.1. Configuration Commands (Type 0 and Type 1)

Because of electrical loading issues, the number of devices that can be supported on a given bus segment is limited. To allow systems to be built beyond a single bus segment, PCI-to-PCI bridges are defined. A PCI-to-PCI bridge requires a mechanism to know how and when to forward configuration accesses to devices that reside behind the bridge.

To support hierarchical PCI buses, two types of configuration transactions are used. They have the formats illustrated in Figure 3-1, which shows the interpretation of **AD** lines during the address phase of a configuration transaction.

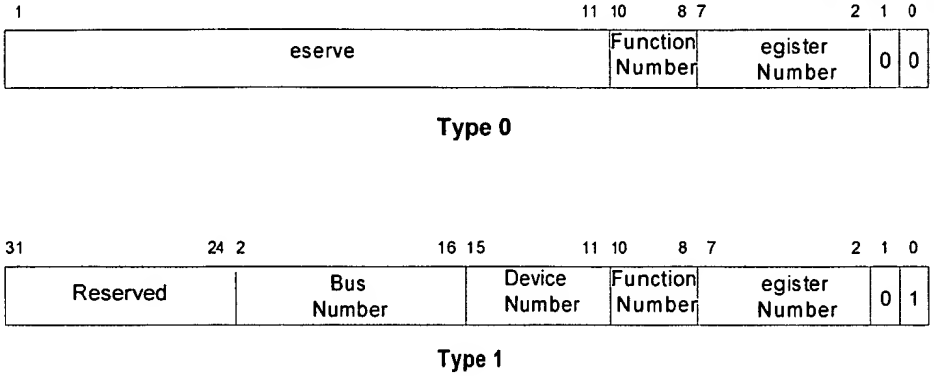



**Figure 3-1: Address Phase Formats of Configuration Transactions**

Type 1 and Type 0 configuration transactions are differentiated by the values on **AD[1::0]**. A Type 0 configuration transaction (when **AD[1::0]** = "00") is used to select a device on the bus where the transaction is being run. A Type 1 configuration transaction (when **AD[1::0]** = "01") is used to pass a configuration request to another bus segment.

The *Register Number* and *Function Number* fields have the same meaning for both configuration types, and *Device Number* and *Bus Number* are used only in Type 1 transactions. Targets must ignore reserved fields.

| | |
|---|---|
| Register Number | is an encoded value used to select a DWORD in the Configuration Space of the intended target. |
| Function Number | is an encoded value used to select one of eight possible functions on a multifunction device. |
| Device Number | is an encoded value used to select one of 32 devices on a given bus. (Refer to Section 3.2.2.3.5. for limitations on the number of devices supported.) |
| Bus Number | is an encoded value used to select 1 of 256 buses in a system. |

Bridges (both host and PCI-to-PCI) that need to generate a Type 0 configuration transaction use the Device Number to select which **IDSEL** to assert. The Function Number is provided on **AD[10::08]**. The Register Number is provided on **AD[7::2]**. **AD[1::0]** must be "00" for a Type 0 configuration transaction.

A Type 0 configuration transaction is not propagated beyond the local PCI bus and must be claimed by a local device or terminated with Master-Abort.

If the target of a configuration transaction resides on another bus (not the local bus), a Type 1 configuration transaction must be used. All targets except PCI-to-PCI bridges ignore Type 1 configuration transactions. PCI-to-PCI bridges decode the Bus Number field to determine if the destination bus of the configuration transaction resides behind the bridge. If the Bus Number is not for a bus behind the bridge, the transaction is ignored. The bridge claims the transaction if the transaction is to a bus behind the bridge. If the Bus Number is not to the secondary bus of the bridge, the transaction is simply passed through unchanged. If the Bus Number matches the secondary bus number, the bridge converts the transaction into a Type 0 configuration transaction. The bridge changes **AD[1::0]** to "00" and passes **AD[10::02]** through unchanged. The

Device Number is decoded to select one of 32 devices on the local bus. The bridge asserts the correct **IDSEL** and initiates a Type 0 configuration transaction. Note: PCI-to-PCI bridges can also forward configuration transactions upstream (refer to the *PCI-to-PCI Bridge Architecture Specification* for more information).

A standard expansion bus bridge must not forward a configuration transaction to an expansion bus.

## 3.2.2.3.2. Software Generation of Configuration Transactions

Systems must provide a mechanism that allows software to generate PCI configuration transactions. This mechanism is typically located in the host bridge. For PC-AT compatible systems, the mechanism[14] for generating configuration transactions is defined and specified in this section. A device driver should use the API provided by the operating system to access the Configuration Space of its device and not directly by way of the hardware mechanism. For other system architectures, the method of generating configuration transactions is not defined in this specification.

Two DWORD I/O locations are used to generate configuration transactions for PC-AT compatible systems. The first DWORD location (CF8h) references a read/write register that is named CONFIG_ADDRESS. The second DWORD address (CFCh) references a read/write register named CONFIG_DATA. The CONFIG_ADDRESS register is 32 bits with the format shown in Figure 3-2. Bit 31 is an enable flag for determining when accesses to CONFIG_DATA are to be translated to configuration transactions on the PCI bus. Bits 30 to 24 are reserved, read-only, and must return 0's when read. Bits 23 through 16 choose a specific PCI bus in the system. Bits 15 through 11 choose a specific device on the bus. Bits 10 through 8 choose a specific function in a device (if the device supports multiple functions). Bits 7 through 2 choose a DWORD in the device's Configuration Space. Bits 1 and 0 are read-only and must return 0's when read.

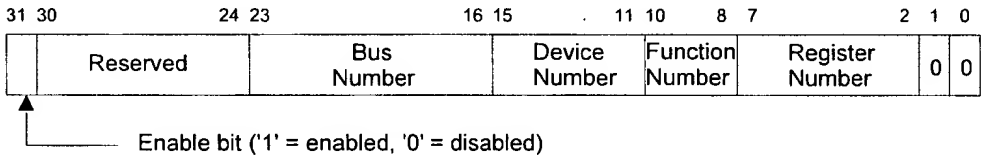



**Figure 3-2: Layout of CONFIG_ADDRESS Register**

Anytime a host bridge sees a full DWORD I/O write from the host to CONFIG_ADDRESS, the bridge must latch the data into its CONFIG_ADDRESS register. On full DWORD I/O reads to CONFIG_ADDRESS, the bridge must return the data in CONFIG_ADDRESS. Any other types of accesses to this address (non-DWORD) have no effect on CONFIG_ADDRESS and are executed as normal I/O transactions on the PCI bus. Therefore, the only I/O Space consumed by this register is a DWORD at the given address. I/O devices that share the same address but use BYTE or WORD registers are not affected because their transactions will pass through the host bridge unchanged.

[14] In versions 2.0 and 2.1 of this specification, two mechanisms were defined. However, only one mechanism (Configuration Mechanism #1) was allowed for new designs and the other (Configuration Mechanism #2) was included for reference.

When a host bridge sees an I/O access that falls inside the DWORD beginning at CONFIG_DATA address, it checks the Enable bit and the Bus Number in the CONFIG_ADDRESS register. If Enable bit is set and the Bus Number matches the bridge's Bus Number or any Bus Number behind the bridge, a configuration cycle translation must be done.

There are two types of translation that take place. The first, Type 0, is a translation where the device being addressed is on the PCI bus connected to the host bridge. The second, Type 1, occurs when the device is on another bus somewhere behind this bridge.

For Type 0 translations (see Figure 3-3), the host bridge does a decode of the Device Number field to assert the appropriate **IDSEL** line[15] and performs a configuration transaction on the PCI bus where **AD[1::0]** = "00". Bits 10 - 8 of CONFIG_ADDRESS are copied to **AD[10::8]** on the PCI bus as an encoded value which is used by components that contain multiple functions. **AD[7::2]** are also copied from the CONFIG_ADDRESS register. Figure 3-3 shows the translation from the CONFIG_ADDRESS register to **AD** lines on the PCI bus.

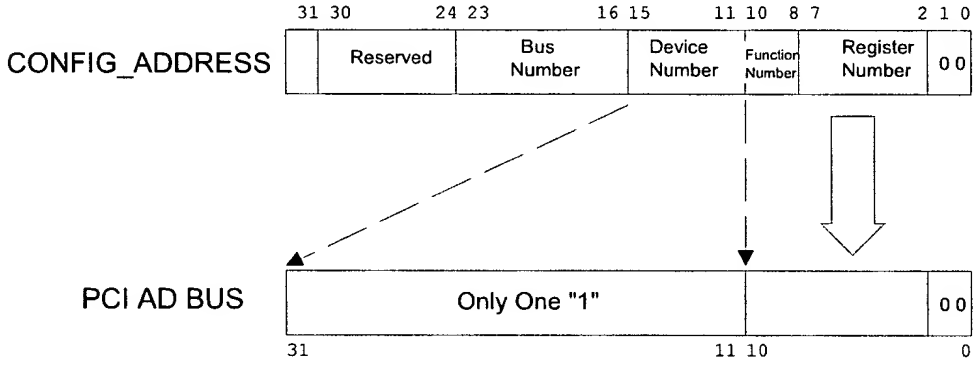



**Figure 3-3: Host Bridge Translation for Type 0 Configuration Transactions Address Phase**

For Type 1 translations, the host bridge directly copies the contents of the CONFIG_ADDRESS register (excluding bits 31 and 0) onto the PCI **AD** lines during the address phase of a configuration transaction making sure that **AD[1::0]** is "01".

In both Type 0 and Type 1 translations, byte enables for the data transfers must be directly copied from the processor bus.

[15] If the Device Number field selects an **IDSEL** line that the bridge does not implement, the bridge must complete the processor access normally, dropping the data on writes and returning all ones on reads. The bridge may optionally implement this requirement by performing a Type 0 configuration access with no **IDSEL** asserted. This will terminate with Master-Abort which drops write data and returns all ones on reads.

**Implementation Note: Bus Numbers Registers and Peer Host Bridges**

For host bridges that do not support peer host buses, translating configuration accesses into configuration transactions is simple. If the Bus Number in the CONFIG_ADDRESS register is zero, a Type 0 configuration translation is used. If the Bus Number in the CONFIG_ADDRESS register is non-zero, a Type 1 configuration translation is used.

For host bridges that support peer host buses, one peer bridge typically is designated to always acknowledge accesses to the CONFIG_ADDRESS register. Other peer bridges would snoop the data written to this register. Accesses to the CONFIG_DATA register are typically handshaken by the bridge doing the configuration translation.

Host bridges that support peer host buses require two Configuration Space registers whose contents are used to determine when the bridge does configuration transaction translation. One register (Bus Number) specifies the bus number of the PCI bus directly behind the bridge, and the other register (Subordinate Bus Number) specifies the number of the last hierarchical bus behind the bridge. (A PCI-to-PCI bridge requires an additional register, which is its Primary Bus Number.) System configuration software is responsible for initializing these registers to appropriate values. The host bridge determines the configuration translation type (1 or 0) based on the value of the bus number in the CONFIG_ADDRESS register. If the Bus Number in the CONFIG_ADDRESS register matches the Bus Number register, a Type 0 configuration transaction is used. If the Bus Number in CONFIG_ADDRESS is greater than the Bus Number register and less than or equal to the Subordinate Bus Number register, a Type 1 configuration transaction is used. If the Bus Number in CONFIG_ADDRESS is less than the Bus Number register or greater than the Subordinate Bus Number register, the configuration transaction is addressing a bus that is not implemented or is behind some other host bridge and is ignored.

## 3.2.2.3.3. Software Generation of Special Cycles

This section defines how a host bridge in a PC-AT compatible systems may optionally implement the configuration mechanism for accessing Configuration Space to allow software to generate a transaction that uses a Special Cycle command. Host bridges are not required to provide a mechanism for allowing software to generate a transaction using a Special Cycle command.

When the CONFIG_ADDRESS register is written with a value such that the Bus Number matches the bridge's bus number, the Device Number is all 1's, the Function Number is all 1's, and the Register Number has a value of zero, then the bridge is primed to generate a transaction using a Special Cycle command the next time the CONFIG_DATA register is written. When the CONFIG_DATA register is written, the bridge generates a transaction that uses a Special Cycle command encoding (rather than Configuration Write command) on the **C/BE[3::0]#** pins during the address phase and drives the data from the I/O write onto **AD[31::00]** during the first data phase. After CONFIG_ADDRESS has been set up this way, reads to CONFIG_DATA have undefined results. In one possible implementation, the bridge can treat it as a normal configuration operation (i.e., generate a Type 0 configuration transaction on the PCI bus). This will terminate with a Master-Abort and the processor will have all 1's returned.

If the Bus Number field of CONFIG_ADDRESS does not match the bridge's bus number, then the bridge passes the write to CONFIG_DATA on through to PCI as a Type 1 configuration transaction just like any other time the bus numbers do not match.

### 3.2.2.3.4. Selection of a Device's Configuration Space

Accesses in the Configuration Address Space require device selection decoding to be done externally and to be signaled to the device via initialization device select, or **IDSEL**, which functions as a classical "chip select" signal. Each device has its own **IDSEL** input (except for host bus bridges, which are permitted to implement their initialization device selection internally).

Devices that respond to Type 0 configuration cycles are separated into two types and are differentiated by an encoding in the Configuration Space header. The first type (single-function device) is defined for backward compatibility and only uses its **IDSEL** pin and **AD[1::0]** to determine whether or not to respond.

A single function device asserts **DEVSEL#** to claim a configuration transaction when:

- a configuration command is decoded;
- the device's **IDSEL** is asserted; and
- **AD[1::0]** is "00" (Type 0 Configuration Command) during the Address Phase.

Otherwise, the device ignores the current transaction. A single-function device may optionally respond to all function numbers as the same function or may decode the Function Number field, **AD[10::08]**, and respond only to function 0 and not respond (Master-Abort termination) to the other function numbers.

The second type of device (multi-function device) decodes the Function Number field **AD[10::08]** to select one of eight possible functions on the device when determining whether or not to respond. Multi-function devices are required to do a full decode on **AD[10::08]** and only respond to the configuration cycle if they have implemented the Configuration Space registers for the selected function. They must not respond (Master-Abort termination) to unimplemented function numbers. They are also required to always implement function 0 in the device. Implementing other functions is optional and may be assigned in any order (i.e., a two-function device must respond to function 0 but can choose any of the other possible function numbers (1-7) for the second function).

If a device implements multiple independent functions, it asserts **DEVSEL#** to claim a configuration transaction when:

- a configuration command is decoded;
- the target's **IDSEL** is asserted;
- **AD[1::0]** is "00"; and
- **AD[10::08]** match a function that is implemented.

Otherwise, the transaction is ignored. For example, if functions 0 and 4 are implemented (functions 1 through 3 and 5 through 7 are not), the device would assert **DEVSEL#** for a configuration transaction in which **IDSEL** is asserted and **AD[1::0]** are 00 and **AD[10::08]** matches 000 or 100. **AD[31::11]** are ignored by a multi-function device during an access of its configuration registers.

The order in which configuration software probes devices residing on a bus segment is not specified. Typically, configuration software either starts with Device Number 0 and works up or starts at Device Number 31 and works down. If a single function device is detected (i.e., bit 7 in the Header Type register of function 0 is 0), no more functions for

that Device Number will be checked. If a multi-function device is detected (i.e., bit 7 in the Header Type register of function 0 is 1), then all remaining Function Numbers will be checked.

Once a function has been selected, it uses **AD[7::2]** to address a DWORD and the byte enables to determine which bytes within the addressed DWORD are being accessed. A function must not restrict the size of the access it supports in Configuration Space. The configuration commands, like other commands, allow data to be accessed using any combination of bytes (including a byte, word, DWORD, or non-contiguous bytes) and multiple data phases in a burst. The target is required to handle any combination of byte enables. However, it is not required to handle a configuration transaction that consists of multiple data phases. If a configuration transaction consists of more than a single data phase, the target is permitted to terminate the request with Disconnect. This is not sufficient cause for the target to terminate the transaction with Target-Abort, since this is not an error condition.

If a configuration transaction has multiple data phases (burst), linear burst ordering is the only addressing mode allowed, since **AD[1::0]** convey configuration transaction type and not a burst addressing mode like Memory accesses. The implied address of each subsequent data phase is one DWORD larger than the previous data phase. For example, a transaction starts with **AD[7::2]** equal to 0000 00xxb, the sequence of a burst would be: 0000 01xxb, 0000 10xxb, 0000 11xxb, 0001 00xxb (where xx indicate whether the transaction is a Type 00 or Type 01 configuration transaction). The rest of the transaction is the same as other commands including all termination semantics.
Note: The *PCI-to-PCI Bridge Architecture Specification* restricts Type 1 configuration transactions that are converted into a transaction that uses a Special Cycle command to a single data phase (no Special Cycle bursts).

If no agent responds to a configuration transaction, the request is terminated via Master-Abort (refer to Section 3.3.3.1.).

## 3.2.2.3.5. System Generation of IDSEL

Exactly how the **IDSEL** pin is driven is left to the discretion of the host/memory bridge or system designer. This signal has been designed to allow its connection to one of the upper 21 address lines, which are not otherwise used in a configuration access. However, there is no specified way of determining **IDSEL** from the upper 21 address bits. Therefore, the **IDSEL** pin must be supported by all targets. Devices must not make an internal connection between an **AD** line and an internal **IDSEL** signal in order to save a pin. The only exception is the host bridge, since it defines how **IDSEL**s are mapped. **IDSEL** generation behind a PCI-to-PCI bridge is specified in the *PCI-to-PCI Bridge Architecture Specification.*

The binding between a device number in the CONFIG_ADDRESS register of PC-AT compatible system and the generation of an **IDSEL** is not specified. Therefore, BIOS must scan all 32 device numbers to ensure all components are located. Note: The hardware that converts the device number to an **IDSEL** is required to ensure that only a single unique **IDSEL** line is asserted for each device number. Configuration transactions that are not claimed by a device are terminated with Master-Abort. The master that initiated this transaction sets the received Master-Abort bit in the Status register.

### Implementation Note: System Generation of IDSEL

How a system generates **IDSEL** is system specific; however, if no other mapping is required, the following example may be used. The **IDSEL** signal associated with Device Number 0 is connected to **AD[16]**, **IDSEL** of Device Number 1 is connected to **AD[17]**, and so forth until **IDSEL** of Device Number 15 is connected to **AD[31]**. For Device Numbers 17-31, the host bridge should execute the transaction but not assert any of the **AD[31::16]** lines but allow the access to be terminated with Master-Abort.

Twenty-one different devices can be uniquely selected for configuration accesses by connecting a different address line to each device and asserting one of the **AD[31::11]** lines at a time. The issue with connecting one of the upper 21 **AD** lines to **IDSEL** is an additional load on the **AD** line. This can be mitigated by resistively coupling **IDSEL** to the appropriate **AD** line. This does, however, create a very slow slew rate on **IDSEL**, causing it to be in an invalid logic state most of the time, as shown in Figure 3-4 with the "XXXX" marks. However, since it is only used on the address phase of a Type 0 configuration transaction, the address bus can be pre-driven a few clocks before **FRAME#**[16], thus guaranteeing **IDSEL** to be stable when it needs to be sampled. Pre-driving the address bus is equivalent to address stepping as discussed in Section 3.6.3. Note that if resistive coupling is used, the bridge that generates the configuration transaction is required to use address stepping or ensure that the clock period is sufficiently long to allow **IDSEL** to become stable before initiating the configuration transaction. For all other cycles, **IDSEL** is undefined and may be at a non-deterministic level during the address phase.

[16] The number of clocks the address bus should be pre-driven is determined from the RC time constant on **IDSEL**.

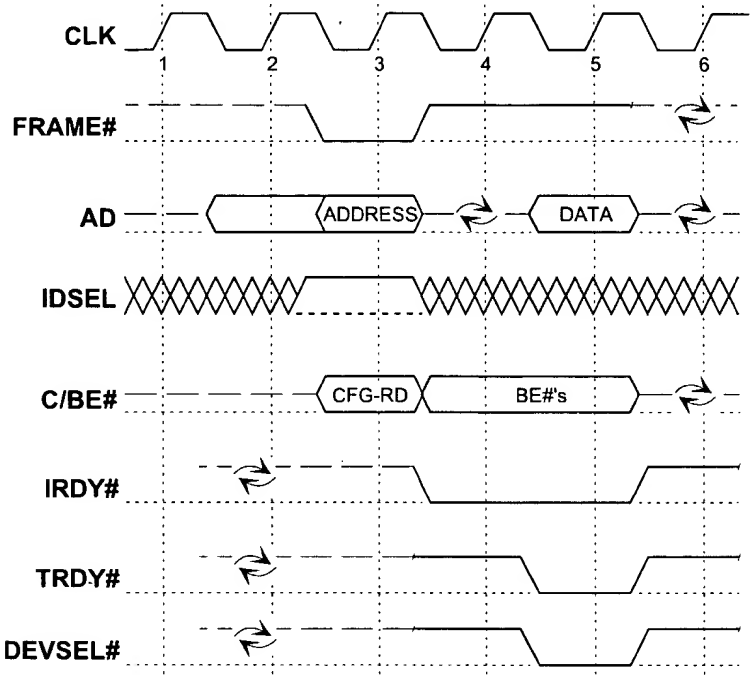



**Figure 3-4: Configuration Read**

## 3.2.3. Byte Lane and Byte Enable Usage

The bus protocol does not support automatic bus sizing on requests of a DWORD or less. (Automatic bus sizing allows a device to request the master of the current transaction to break the access into smaller pieces for the target to complete. For example, an 8-bit device that is accessed with a 16-bit request could transfer the lower 8 bits and require the master to move the upper 8 bits (of the 16-bit access) down to the lower byte lane to complete the request.) Since all PCI devices connect to the lower 32 bits for address decode, the device itself is required to provide this byte steering when required, or the driver is required to place the data on the correct byte. In general, software is aware of the characteristics of the target device and only issues appropriate length accesses.

The bus protocol requires automatic bus sizing if a master requests a 64-bit data transfer to a 32-bit target. In this case, the target does not indicate that it can do a 64-bit data transfer, and the master is required to complete the current transaction using 32-bit data transfers. For more details about 64-bit data transactions, refer to Section 3.8.

The byte enables alone are used to determine which byte lanes carry meaningful data. The byte enables are free to change between data phases but must be valid on the clock that starts each data phase and must stay valid for the entire data phase. In Figure 3-5, data phases begin on clocks 3, 5, and 7. (Changing byte enables during a read burst transaction is generally not useful, but is permitted.) The master is free to change the byte enables on each new data phase (although the read diagram does not show this). If the master changes byte enables on a read transaction, it does so with the same timing as would be used in a write transaction. If byte enables are important for the target on a read transaction, the target must wait for the byte enables to be valid on each data phase

before completing the transfer; otherwise, it must return all bytes. Note: Byte enables are valid during the entire data phase independent of the state of **IRDY#**.

If a target supports prefetching (bit 3 is set in the Memory Base Address register -- refer to Section 6.2.5.1.), it must also return all data[17] regardless of which byte enables are asserted. A target can only operate in this mode when there are no side effects (data destroyed or status changes because of the access).

PCI allows any contiguous or non-contiguous combination of byte enables. If no byte enables are asserted, the target of the access must complete the data phase by asserting **TRDY#** and providing parity if the transaction is a read request. The target of an access where no byte enables are asserted must complete the current data phase without any state change. On a read transaction, this means that data and status are not changed. If completing the access has no affect on the data or status, the target may complete the access by either providing data or not. The generation and checking of parity is the same regardless of the state of the byte enables for both 32-bit and 64-bit data transfers. For a discussion on parity generation and checking, refer to Section 3.7.1. (32-bit transactions) and Section 3.8. (64-bit transactions).

However, some targets may not be able to properly interpret non-contiguous patterns (e.g., expansion bus bridges that interface to 8- and 16-bit devices). Expansion bus bridges may optionally report patterns that are illegal on the expansion bus as an asynchronous error (**SERR#**) or break the transaction into smaller transactions that are legal for the intended agent. The target of an I/O transaction is required to signal Target-Abort if it is unable to complete the entire access defined by the byte enables.

## 3.2.4. Bus Driving and Turnaround

A turnaround cycle is required on all signals that are driven by more than one agent. The turnaround cycle is required to avoid contention when one agent stops driving a signal and another agent begins driving the signal. This is indicated on the timing diagrams as two arrows pointing at each others' tail. This turnaround cycle occurs at different times for different signals. For instance, **IRDY#**, **TRDY#**, **DEVSEL#**, **STOP#**, and **ACK64#** use the address phase as their turnaround cycle. **FRAME#**, **REQ64#**, **C/BE[3::0]#**, **C/BE[7::4]#**, **AD[31::00]**, and **AD[63::32]** use the Idle state between transactions as their turnaround cycle. The turnaround cycle for **LOCK#** occurs one clock after the current owner releases it. **PERR#** has a turnaround cycle on the fourth clock after the last data phase, which is three clocks after the turnaround-cycle for the **AD** lines. An Idle state is when both **FRAME#** and **IRDY#** are deasserted (e.g., clock 9 in Figure 3-5).

All **AD** lines (including **AD[63::32]** when the master supports a 64-bit data path) must be driven to stable values during 64-bit transfers every address and data phase. Even byte lanes not involved in the current data transfer must physically drive stable (albeit meaningless) data onto the bus. The motivation is for parity calculations and to keep input buffers on byte lanes not involved in the transfer from switching at the threshold level and, more generally, to facilitate fast metastability-free latching. In power-sensitive applications, it is recommended that in the interest of minimizing bus switching power consumption, byte lanes not being used in the current bus phase should be driven with the same data as contained in the previous bus phase. In applications that are not power

[17] For a 32-bit data transfer, this means 4 bytes per data phase; for a 64-bit data transfer, this means 8 bytes per data phase.

sensitive, the agent driving the **AD** lines may drive whatever it desires on unused byte lanes. Parity must be calculated on all bytes regardless of the byte enables.

## 3.2.5. Transaction Ordering and Posting

Transaction ordering rules on PCI accomplish three things. First, they satisfy the write-results ordering requirements of the Producer-Consumer Model. This means that the results of writes from one master (the Producer) anywhere in the system are observable by another master (the Consumer) anywhere in the system only in their original order. (Different masters (Producers) in different places in the system have no fundamental need for their writes to happen in a particular order with respect to each other, since each will have a different Consumer. In this case, the rules allow for some writes to be rearranged.) Refer to Appendix E for a complete discussion of the Producer-Consumer Model. Second, they allow for some transactions to be posted to improve performance. And third, they prevent bus deadlock conditions, when posting buffers have to be flushed to meet the first requirement.

The order relationship of a given transaction with respect to other transactions is determined when it completes; i.e., when data is transferred. Transactions which terminate with Retry have not completed since no data was transferred and, therefore, have no ordering requirements relative to each other. Transactions that terminate with Master-Abort or Target-Abort are considered completed with or without data being transferred and will not be repeated by the master. The system may accept requests in any order, completing one while continuing to Retry another. If a master requires one transaction to be completed before another, the master must not attempt the second transaction until the first one is complete. If a master has only one outstanding request at a time, then that master's transactions will complete throughout the system in the same order the master executed them. Refer to Section 3.3.3.3.5.3. for further discussion of request ordering.

Transactions can be divided into two general groups based on how they are handled by an intermediate agent, such as a bridge. The two groups are posted and non-posted transactions. Posted transactions complete at the originating device before they reach their ultimate destination. The master will often proceed with other work, sometimes including other bus transactions, before the posted transaction reaches it ultimate destination. In essence, the intermediate agent of the access (e.g., a bridge) accepts the data on behalf of the actual target and assumes responsibility for ensuring that the access completes at the final destination. Memory writes (Memory Write and Memory Write and Invalidate commands) are allowed to be posted on the PCI bus.

Non-posted transactions reach their ultimate destination before completing at the originating device. The master cannot proceed with any other work until the transaction has completed at the ultimate destination (if a dependency exists). Memory read transactions (Memory Read, Memory Read Line, and Memory Read Multiple), I/O transactions (I/O Read and I/O Write), and configuration transactions (Configuration Read and Configuration Write) are non-posted (except as noted below for host bridges).

There are two categories of devices with different requirements for transaction ordering and posting. Each category will be presented separately.

### 3.2.5.1. Transaction Ordering and Posting for Simple Devices

A simple device is any device that while acting as a bus master does not require its write data to be posted at the bus interface logic. Generally devices that do not connect to local CPUs are implemented as simple devices.

The target and master state machines in the PCI interface of a simple device are completely independent. A simple device cannot make the completion of any transaction (either posted or non-posted) as a target contingent upon the prior completion of any other transaction as a master. Simple devices are allowed to terminate a transaction with Retry only to execute the transaction as a Delayed Transaction or for temporary conditions which are guaranteed to be resolved with time; e.g., during a video screen refresh or while a transaction buffer is filled with transactions moving in the same direction. (Refer to Section 3.5.3. for a limit on the length of time a memory write transaction can be terminated with Retry.)

**Implementation Note: Deadlock When Target and Master Not Independent**

The following is an example of a deadlock that could occur if devices do not make their target and master interfaces independent.

Suppose two devices, Device A and Device B, are talking directly to each other. Both devices attempt I/O writes to each other simultaneously. Suppose Device A is granted the bus first and executes its I/O write addressing Device B. Device B decodes its address and asserts **DEVSEL#**. Further, suppose that Device B violates the requirement for the target state machine to be independent of the master state machine and always terminates transactions as a target with Retry until its master state machine completes its outstanding requests. Since Device B also has an I/O transaction it must execute as a master, it terminates Device A's transaction with Retry.

Device B is then granted the bus, and Device B executes its I/O write addressing Device A. If Device A responds the same way Device B did, the system will deadlock.

**Implementation Note: Deadlock When Posted Write Data is Not Accepted**

Deadlocks can also occur when a device does not accept a memory write transaction from a bridge. As described below, a bridge is required in certain cases to flush its posting buffer as a master before it completes a transaction as a target. Suppose a PCI-to-PCI bridge contains posted memory write data addressed to a downstream device. But before the bridge can acquire the downstream bus to do the write transaction, a downstream device initiates a read from host memory. Since requirement 3 in the bridge rules presented below states that posting buffers must be flushed before a read transaction can be completed, the bridge must Retry the agent's read and attempt a write transaction. If the downstream device were to make the acceptance of the write data contingent upon the prior completion of the retried read transaction (that is, if it could not accept the posted write until it first completed the read transaction), the bus would be deadlocked.

Since certain PCI-to-PCI bridge devices designed to previous versions of this specification require their posting buffer to be flushed before starting any non-posted transaction, the same deadlock could occur if the downstream device makes the acceptance of a posted write contingent on the prior completion of any non-posted transaction.

The required independence of target and master state machines in a simple device implies that a simple device cannot internally post any outbound transactions. For example, if during the course of performing its intended function a device must execute a memory write as a master on the PCI bus, the device cannot post that memory write in the master interface of the device. More specifically, the device cannot proceed to other internal operations such as updating status registers that would be observable by another master in the system. The simple device must wait until the memory write transaction completes on the PCI bus (**TRDY#** asserted; Master-Abort or Target-Abort) before proceeding internally.

Simple devices are strongly encouraged to post inbound memory write transactions to speed the transaction on the PCI bus. How such a device deals with ordering of inbound posted write data is strictly implementation dependent and beyond the scope of this specification.

Simple devices do not support exclusive accesses and do not use the **LOCK#** signal. Refer to Appendix F for a discussion of the use of **LOCK#** in bridge devices.

## 3.2.5.2. Transaction Ordering and Posting for Bridges

A bridge device is any device that implements internal posting of outbound memory write transactions, i.e., write transactions that must be executed by the device as a master on the PCI bus. Bridges normally join two buses such as two PCI buses, a host bus and a PCI bus, or a PCI bus and a bus for a local CPU; i.e., a peripheral CPU.

Bridges are permitted to post memory write transactions moving in either direction through the bridge. The following ordering rules guarantee that the results of one master's write transactions are observable by other masters in the proper order, even though the write transaction may be posted in a bridge. They also guarantee that the bus does not deadlock when a bridge tries to empty its posting buffers.

- Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus. Even if a single burst on the originating bus is terminated with Disconnect on the destination bus so that it is broken into multiple transactions, those transactions must not allow the data phases to complete on the destination bus in any order other than their order on the originating bus.
- Write transactions flowing in one direction through a bridge have no ordering requirements with respect to writes flowing in the other direction through the bridge.
- Posted memory write buffers in both directions must be flushed before completing a read transaction in either direction. Posted memory writes originating on the *same* side of the bridge as a read transaction, and completing before the read command completes on the originating bus, must complete on the destination bus in the same order. Posted memory writes originating on the opposite side of the bridge from a read transaction and completing on the read-destination bus before the read command completes on the read-destination bus must complete on the read-origin bus in the same order. In other words, a read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. In addition, before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.
- A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of a locked transaction as a master only if the bridge has already established a locked operation with its intended target; otherwise, a deadlock may occur. (Refer to Appendix F for a discussion of the use of **LOCK#** in bridge devices.) In all other cases, bridges are allowed to refuse to accept a memory write only for temporary conditions which are guaranteed to be resolved with time, e.g., during a video screen refresh or while the memory buffer is filled by previous memory write transactions moving in the same direction.

Host bus bridges are permitted to post I/O write transactions that originate on the host bus and complete on a PCI bus segment when they follow the ordering rules described in this specification and do not cause a deadlock. This means that when a host bus bridge posts an I/O write transaction that originated on the host bus, it must provide a deadlock free environment when the transaction completes on PCI. The transaction will complete on the destination PCI bus before completing on the originating PCI bus.

Since memory write transactions may be posted in bridges anywhere in the system, and I/O writes may be posted in the host bus bridge, a master cannot automatically tell when its write transaction completes at the final destination. For a device driver to guarantee that a write has completed at the actual target (and not at an intermediate bridge), it must complete a read to the same device that the write targeted. The read (memory or I/O) forces all bridges between the originating master and the actual target to flush all posted

data before allowing the read to complete. For additional details on device drivers, refer to Section 6.5. Refer to Section 3.10., item 6, for other cases where a read is necessary.

Interrupt requests (that use **INTx#**) do not appear as transactions on the PCI bus (they are sideband signals) and, therefore, have no ordering relationship to any bus transactions. Furthermore, the system is not required to use the Interrupt Acknowledge bus transaction to service interrupts. So interrupts are not synchronizing events and device drivers cannot depend on them to flush posting buffers. However, when MSI are used, they have the same ordering rules as a memory write transaction (refer to Section 6.8. for more information).

## 3.2.6. Combining, Merging, and Collapsing

Under certain conditions, bridges that receive (write) data may attempt to convert a transaction (with a single or multiple data phases) into a larger transaction to optimize the data transfer on PCI. The terms used when describing the action are: combining, merging, and collapsing. Each term will be defined and the usage for bridges (host, PCI-to-PCI, or standard expansion bus) will be discussed.

**Combining** -- occurs when sequential memory write transactions (single data phase or burst and independent of active byte enables) are combined into a single PCI bus transaction (using linear burst ordering).

The combining of data is not required but is recommended whenever posting of write data is being done. Combining is permitted only when the implied ordering is not changed. Implied ordering means that the target sees the data in the same order as the original master generated it. For example, a write sequence of DWORD 1, 2, and 4 can be converted into a burst sequence. However, a write of DWORD 4, 3, and 1 cannot be combined into a burst but must appear on PCI as three separate transactions in the same order as they occurred originally. Bursts may include data phases that have no byte enables asserted. For example, the sequence DWORD 1, 2, and 4 could be combined into a burst in which data phase 1 contains the data and byte enables provided with DWORD 1. The second data phase of the burst uses data and byte enables provided with DWORD 2, while data phase 3 asserts no byte enables and provides no meaningful data. The burst completes with data phase 4 using data and byte enables provided with DWORD 4.

If the target is unable to handle multiple data phases for a single transaction, it terminates the burst transaction with Disconnect with or after each data phase. The target sees the data in the same order the originating master generated it, whether the transaction was originally generated as a burst or as a series of single data phase accesses which were combined into a burst.

**Byte Merging** -- occurs when a sequence of individual memory writes (bytes or words) are merged into a single DWORD.

The merging of bytes within the same DWORD for 32-bit transfers or QUADWORD (eight bytes) for 64-bit transfers is not required but is recommended when posting of write data is done. Byte merging is permitted only when the bytes within a data phase are in a prefetchable address range. While similar to combining in concept, merging can be done in any order (within the same data phase) as long as each byte is only written once. For example, in a sequence where bytes 3, 1, 0, and 2 are written to the same DWORD address, the bridge could merge them into a single data phase memory write on PCI with Byte Enable 0, 1, 2, and 3 all asserted instead of four individual write transactions. However, if the sequence written to the same DWORD address were

byte 1, and byte 1 again (with the same or different data), byte 2, and byte 3, the bridge cannot merge the first two writes into a single data phase because the same byte location must be written twice. However, the last three transactions could be merged into a single data phase with Byte Enable 0 being deasserted and Byte Enable 1, 2, and 3 being asserted. Merging can never be done to a range of I/O or Memory Mapped I/O addresses (not prefetchable).

Note: Merging and combining can be done independently of each other. Bytes within a DWORD may be merged and merged DWORDs can be combined with other DWORDs when conditions allow. A device can implement only byte merging, only combining, both byte merging and combining, or neither byte merging or combining.

**Collapsing** -- is when a sequence of memory writes to the same location (byte, word, or DWORD address) are collapsed into a single bus transaction.

Collapsing is not permitted by PCI bridges (host, PCI-to-PCI, or standard expansion) except as noted below. For example, a memory write transaction with Byte Enable 3 asserted to DWORD address X, followed by a memory write access to the same address (X) as a byte, word, or DWORD, or any other combination of bytes allowed by PCI where Byte Enable 3 is asserted, cannot be merged into a single PCI transaction. These two accesses must appear on PCI as two separate and distinct transactions.

Note: The combining and merging of I/O and Configuration transactions are not allowed. The collapsing of data of any type of transaction (Configuration, Memory, or I/O) is never allowed (except where noted below).

Note: If a device cannot tolerate memory write combining, it has been designed incorrectly. If a device cannot tolerate memory write byte merging, it must mark itself as not prefetchable. (Refer to Section 6.2.5.1. for a description of prefetchable.) A device that marks itself prefetchable must tolerate combining (without reordering) and byte merging (without collapsing) of writes as described previously. A device is explicitly not required to tolerate reordering of DWORDs or collapsing of data. A prefetchable address range may have write side effects, but it may not have read side effects. A bridge (host bus, PCI-to-PCI, or standard expansion bus) cannot reorder DWORDs in any space, even in a prefetchable space.

Bridges may optionally allow data to be collapsed in a specific address range when a device driver indicates that there are no adverse side-effects due to collapsing. How a device driver indicates this to the system is beyond the scope of this specification.

**Implementation Note: Combining, Merging, and Collapsing**

Bridges that post memory write data should consider implementing Combining and Byte Merging. The collapsing of multiple memory write transactions into a single PCI bus transaction is never allowed (except as noted above). The combining of sequential DWORD memory writes into a PCI burst has significant performance benefits. For example, a processor is doing a large number of DWORD writes to a frame buffer. When the host bus bridge combines these accesses into a single PCI transaction, the PCI bus can keep up with a host bus that is running faster and/or wider than PCI.

The merging of bytes within a single DWORD provides a performance improvement but not as significant as combining. However, for unaligned multi-byte data transfers merging allows the host bridge to merge misaligned data into single DWORD memory write transactions. This reduces (at a minimum) the number of PCI transactions by a factor of two. When the bridge merges bytes into a DWORD and then combines DWORDs into a burst, the number of transactions on PCI can be reduced even further than just by merging. With the addition of combining sequential DWORDs, the number of transactions on PCI can be reduced even further. Merging data (DWORDs) within a single cacheline appears to have minimal performance gains.

## 3.3. Bus Transactions

The timing diagrams in this section show the relationship of significant signals involved in 32-bit transactions. When a signal is drawn as a solid line, it is actively being driven by the current master or target. When a signal is drawn as a dashed line, no agent is actively driving it. However, it may still be assumed to contain a stable value if the dashed line is at the high rail. Tri-stated signals are indicated to have indeterminate values when the dashed line is between the two rails (e.g., **AD** or **C/BE#** lines). When a solid line becomes a dotted line, it indicates the signal was actively driven and now is tri-stated. When a solid line makes a low to high transition and then becomes a dotted line, it indicates the signal was actively driven high to precharge the bus and then tri-stated. The cycles before and after each transaction will be discussed in Section 3.4.

## 3.3.1. Read Transaction

Figure 3-5 illustrates a read transaction and starts with an address phase which occurs when **FRAME#** is asserted for the first time and occurs on clock 2. During the address phase, **AD[31::00]** contain a valid address and **C/BE[3::0]#** contain a valid bus command.

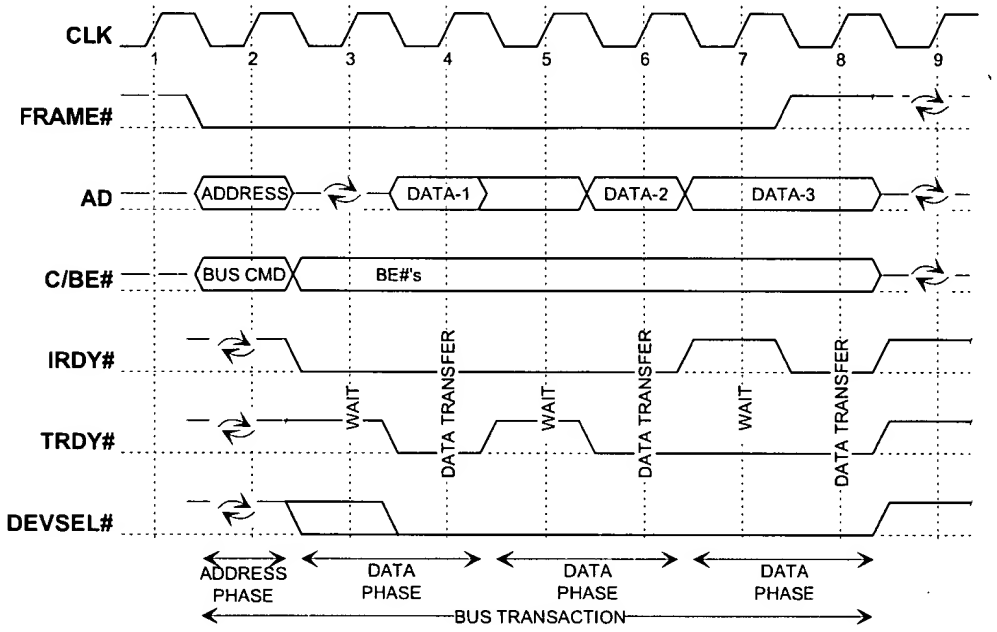



**Figure 3-5: Basic Read Operation**

The first clock of the first data phase is clock 3. During the data phase, **C/BE#** indicate which byte lanes are involved in the current data phase. A data phase may consist of wait cycles and a data transfer. The **C/BE#** output buffers must remain enabled (for both read and writes) from the first clock of the data phase through the end of the transaction. This ensures **C/BE#** are not left floating for long intervals. The **C/BE#** lines contain valid byte enable information during the entire data phase independent of the state of **IRDY#**. The **C/BE#** lines contain the byte enable information for data phase N+1 on the clock following the completion of the data phase N. This is not shown in Figure 3-5 because a burst read transaction typically has all byte enables asserted; however, it is shown in Figure 3-6. Notice on clock 5 in Figure 3-6, the master inserted a wait state by deasserting **IRDY#**. However, the byte enables for data phase 3 are valid on clock 5 and remain valid until the data phase completes on clock 8.

The first data phase on a read transaction requires a turnaround-cycle (enforced by the target via **TRDY#**). In this case, the address is valid on clock 2 and then the master stops driving **AD**. The earliest the target can provide valid data is clock 4. The target must drive the **AD** lines following the turnaround cycle when **DEVSEL#** is asserted. Once enabled, the output buffers must stay enabled through the end of the transaction. (This ensures that the **AD** lines are not left floating for long intervals.)

One way for a data phase to complete is when data is transferred, which occurs when both **IRDY#** and **TRDY#** are asserted on the same rising clock edge. There are other conditions that complete a data phase and these are discussed in Section 3.3.3.2.

(**TRDY#** cannot be driven until **DEVSEL#** is asserted.) When either **IRDY#** or **TRDY#** is deasserted, a wait cycle is inserted and no data is transferred. As noted in Figure 3-5, data is successfully transferred on clocks 4, 6, and 8 and wait cycles are inserted on clocks 3, 5, and 7. The first data phase completes in the minimum time for a read transaction. The second data phase is extended on clock 5 because **TRDY#** is deasserted. The last data phase is extended because **IRDY#** was deasserted on clock 7.

The master knows at clock 7 that the next data phase is the last. However, because the master is not ready to complete the last transfer (**IRDY#** is deasserted on clock 7), **FRAME#** stays asserted. Only when **IRDY#** is asserted can **FRAME#** be deasserted as occurs on clock 8, indicating to the target that this is the last data phase of the transaction.

## 3.3.2. Write Transaction

Figure 3-6 illustrates a write transaction. The transaction starts when **FRAME#** is asserted for the first time which occurs on clock 2. A write transaction is similar to a read transaction except no turnaround cycle is required following the address phase because the master provides both address and data. Data phases work the same for both read and write transactions.

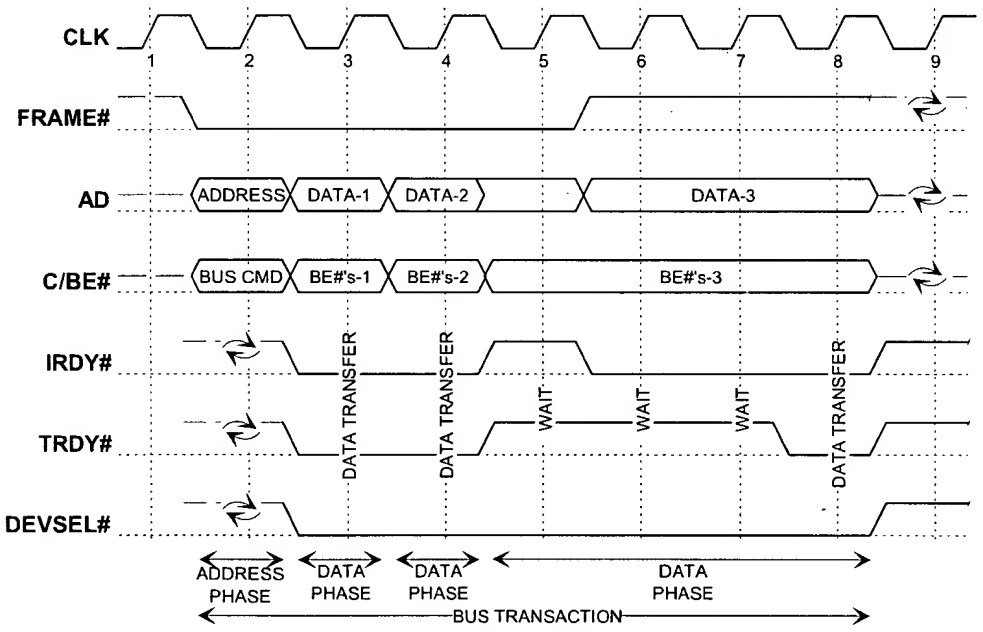



**Figure 3-6: Basic Write Operation**

In Figure 3-6, the first and second data phases complete with zero wait cycles. However, the third data phase has three wait cycles inserted by the target. Notice both agents insert a wait cycle on clock 5. **IRDY#** must be asserted when **FRAME#** is deasserted indicating the last data phase.

The data transfer was delayed by the master on clock 5 because **IRDY#** was deasserted. The last data phase is signaled by the master on clock 6, but it does not complete until clock 8.

Note: Although this allowed the master to delay data, it did not allow the byte enables to be delayed.

## 3.3.3. Transaction Termination

Termination of a PCI transaction may be initiated by either the master or the target. While neither can actually stop the transaction unilaterally, the master remains in ultimate control, bringing all transactions to an orderly and systematic conclusion regardless of what caused the termination. All transactions are concluded when **FRAME#** and **IRDY#** are both deasserted, indicating an Idle state (e.g., clock 9 in Figure 3-6).

### 3.3.3.1. Master Initiated Termination

The mechanism used in master initiated termination is when **FRAME#** is deasserted and **IRDY#** is asserted. This condition signals the target that the final data phase is in progress. The final data transfer occurs when both **IRDY#** and **TRDY#** are asserted. The transaction reaches completion when both **FRAME#** and **IRDY#** are deasserted (Idle state).

The master may initiate termination using this mechanism for one of two reasons:

*Completion* refers to termination when the master has concluded its intended transaction. This is the most common reason for termination.

*Timeout* refers to termination when the master's **GNT#** line is deasserted and its internal Latency Timer has expired. The intended transaction is not necessarily concluded. The timer may have expired because of target-induced access latency or because the intended operation was very long. Refer to Section 3.5.4. for a description of the Latency Timer operation.

A Memory Write and Invalidate transaction is not governed by the Latency Timer except at cacheline boundaries. A master that initiates a transaction with the Memory Write and Invalidate command ignores the Latency Timer until a cacheline boundary. When the transaction reaches a cacheline boundary and the Latency Timer has expired (and **GNT#** is deasserted), the master must terminate the transaction.

A modified version of this termination mechanism allows the master to terminate the transaction when no target responds. This abnormal termination is referred to as *Master-Abort*. Although it may cause a fatal error for the application originally requesting the transaction, the transaction completes gracefully, thus preserving normal PCI operation for other agents.

Two examples of normal completion are shown in Figure 3-7. The final data phase is indicated by the deassertion of **FRAME#** and the assertion of **IRDY#**. The final data phase completes when **FRAME#** is deasserted and **IRDY#** and **TRDY#** are both asserted. The bus reaches an Idle state when **IRDY#** is deasserted, which occurs on clock 4. Because the transaction has completed, **TRDY#** is deasserted on clock 4 also. Note: **TRDY#** is not required to be asserted on clock 3, but could have delayed the final data transfer (and transaction termination) until it is ready by delaying the final assertion

of **TRDY#**. If the target does that, the master is required to keep **IRDY#** asserted until the final data transfer occurs.

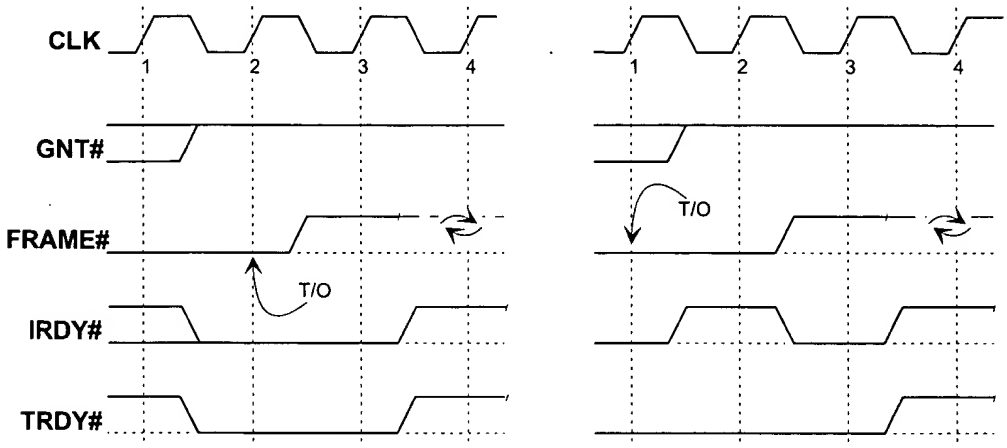



**Figure 3-7: Master Initiated Termination**

Both sides of Figure 3-7 could have been caused by a timeout termination. On the left side, **FRAME#** is deasserted on clock 3 because the timer expires, **GNT#** is deasserted, and the master is ready (**IRDY#** asserted) for the final transfer. Because **GNT#** was deasserted when the timer expired, continued use of the bus is not allowed except when using the Memory Write and Invalidate command (refer to Section 3.5.4.), which must be stopped at the cacheline boundary. Termination then proceeds as normal. If **TRDY#** is deasserted on clock 2, that data phase continues until **TRDY#** is asserted. **FRAME#** must remain deasserted and **IRDY#** must remain asserted until the data phase completes.

The right-hand example shows a timer expiring on clock 1. Because the master is not ready to transfer data (**IRDY#** is deasserted on clock 2), **FRAME#** is required to stay asserted. **FRAME#** is deasserted on clock 3 because the master is ready (**IRDY#** is asserted) to complete the transaction on clock 3. The master must be driving valid data (write) or be capable of receiving data (read) whenever **IRDY#** is asserted. This delay in termination should not be extended more than two or three clocks. Also note that the transaction need not be terminated after timer expiration unless **GNT#** is deasserted.

Master-Abort termination, as shown in Figure 3-8, is an abnormal case (except for configuration or Special Cycle commands) of master initiated termination. A master determines that there will be no response to a transaction if **DEVSEL#** remains deasserted on clock 6. (For a complete description of **DEVSEL#** operation, refer to Section 3.6.1.) The master must assume that the target of the access is incapable of dealing with the requested transaction or that the address was bad and must not repeat the transaction. Once the master has detected the missing **DEVSEL#** (clock 6 in this example), **FRAME#** is deasserted on clock 7 and **IRDY#** is deasserted on clock 8. The earliest a master can terminate a transaction with Master-Abort is five clocks after **FRAME#** was first sampled asserted, which occurs when the master attempts a single data transfer. If a burst is attempted, the transaction is longer than five clocks. However, the master may take longer to deassert **FRAME#** and terminate the access. The master must support the **FRAME#** -- **IRDY#** relationship on all transactions including Master-Abort. **FRAME#** cannot be deasserted before **IRDY#** is asserted, and **IRDY#** must

remain asserted for at least one clock after **FRAME#** is deasserted even when the transaction is terminated with Master-Abort.

Alternatively, **IRDY#** could be deasserted on clock 7, if **FRAME#** was deasserted as in the case of a transaction with a single data phase. The master will normally not repeat a transaction terminated with Master-Abort. (Refer to Section 3.7.4.) Note: If **DEVSEL#** had been asserted on clocks 3, 4, 5, or 6 of this example, it would indicate the request had been acknowledged by an agent and Master-Abort termination would not be permissible.

The host bus bridge, in PC compatible systems, must return all 1's on a read transaction and discard data on a write transaction when terminated with Master-Abort. The bridge is required to set the Master-Abort detected bit in the status register. Other master devices may report this condition as an error by signaling **SERR#** when the master cannot report the error through its device driver. A PCI-to-PCI bridge must support PC compatibility as described for the host bus bridge. When the PCI-to-PCI bridge is used in other systems, the bridge behaves like other masters and reports an error. Prefetching of read data beyond the actual request by a bridge must be totally transparent to the system. This means that when a prefetched transaction is terminated with Master-Abort, the bridge must simply stop the transaction and continue normal operation without reporting an error. This occurs when a transaction is not claimed by a target.

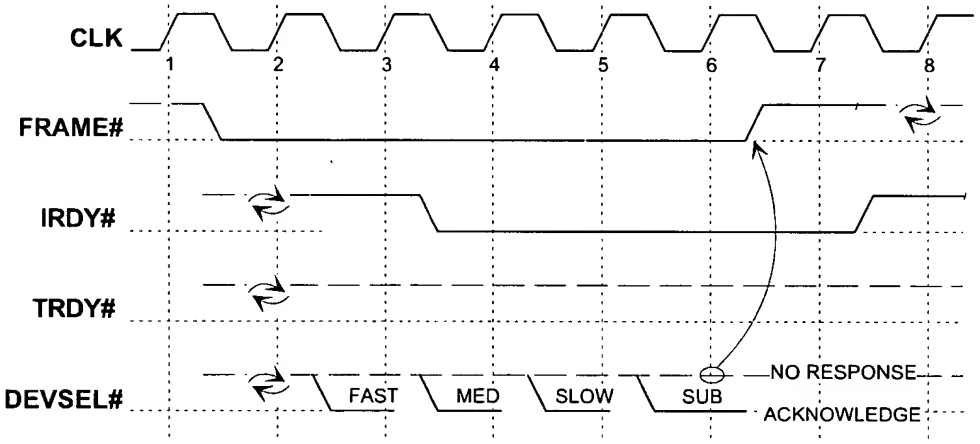



**Figure 3-8: Master-Abort Termination**

In summary, the following general rules govern **FRAME#** and **IRDY#** in all PCI transactions:

1. **FRAME#** and its corresponding **IRDY#** define the Busy/Idle state of the bus; when either is asserted, the bus is Busy; when both are deasserted, the bus is Idle.
2. Once **FRAME#** has been deasserted, it cannot be reasserted during the same transaction.
3. **FRAME#** cannot be deasserted unless **IRDY#** is asserted. (**IRDY#** must always be asserted on the first clock edge that **FRAME#** is deasserted.)
4. Once a master has asserted **IRDY#**, it cannot change **IRDY#** or **FRAME#** until the current data phase completes.
5. The master must deassert **IRDY#** the clock after the completion of the last data phase.

### 3.3.3.2. Target Initiated Termination

Under most conditions, the target is able to source or sink the data requested by the master until the master terminates the transaction. But when the target is unable to complete the request, it may use the **STOP#** signal to initiate termination of the transaction. How the target combines **STOP#** with other signals will indicate to the master something about the condition which lead to the termination.

The three types of target initiated termination are:

*Retry* refers to termination requested before any data is transferred because the target is busy and temporarily unable to process the transaction. This condition may occur, for example, because the device cannot meet the initial latency requirement, is currently locked by another master, or there is a conflict for a internal resource.

Retry is a special case of Disconnect without data being transferred on the initial data phase.

The target signals Retry by asserting **STOP#** and not asserting **TRDY#** on the initial data phase of the transaction (**STOP#** cannot be asserted during the turn-around cycle between the address phase and first data phase of a read transaction). When the target uses Retry, no data is transferred.

*Disconnect* refers to termination requested with or after data was transferred on the initial data phase because the target is unable to respond within the target subsequent latency requirement and, therefore, is temporarily unable to continue bursting. This might be because the burst crosses a resource boundary or a resource conflict occurs. Data may or may not transfer on the data phase where Disconnect is signaled. Notice that Disconnect differs from Retry in that Retry is always on the initial data phase, and no data transfers. If data is transferred with or before the target terminates the transaction, it is a Disconnect. This may also occur on the initial data phase because the target is not capable of doing a burst.

Disconnect with data may be signaled on any data phase by asserting **TRDY#** and **STOP#** together. This termination is used when the target is only willing to complete the current data phase and no more.

Disconnect without data may be signaled on any subsequent data phase (meaning data was transferred on the previous data phase) by deasserting **TRDY#** and asserting **STOP#**.

*Target-Abort* refers to an abnormal termination requested because the target detected a fatal error or the target will never be able to complete the request. Although it may cause a fatal error for the application originally requesting the transaction, the transaction completes gracefully, thus, preserving normal operation for other agents. For example, a master requests all bytes in an I/O Address Space DWORD to be read, but the target design restricts access to a single byte in this range. Since the target cannot complete the request, the target terminates the request with Target-Abort.

Once the target has claimed an access by asserting **DEVSEL#**, it can signal Target-Abort on any subsequent clock. The target signals Target-Abort by deasserting **DEVSEL#** and asserting **STOP#** at the same time.

Most targets will be required to implement at least Retry capability, but any other versions of target initiated termination are optional for targets. Masters must be capable of properly dealing with them all. Retry is optional to very simple targets that:

- do not support exclusive (locked) accesses
- do not have a posted memory write buffer which needs to be flushed to meet the PCI ordering rules
- cannot get into a state where they may need to reject an access
- can always meet target initial latency

A target is permitted to signal Disconnect with data (assert **STOP#** and **TRDY#)** on the initial data phase even if the master is not bursting; i.e., **FRAME#** is deasserted.

### 3.3.3.2.1. Target Termination Signaling Rules

The following general rules govern **FRAME#**, **IRDY#**, **TRDY#**, **STOP#**, and **DEVSEL#** while terminating transactions.

1. A data phase completes on any rising clock edge on which **IRDY#** is asserted and either **STOP#** or **TRDY#** is asserted.
2. Independent of the state of **STOP#**, a data transfer takes place on every rising edge of clock where both **IRDY#** and **TRDY#** are asserted.
3. Once the target asserts **STOP#**, it must keep **STOP#** asserted until **FRAME#** is deasserted, whereupon it must deassert **STOP#**.
4. Once a target has asserted **TRDY#** or **STOP#**, it cannot change **DEVSEL#**, **TRDY#**, or **STOP#** until the current data phase completes.
5. Whenever **STOP#** is asserted, the master must deassert **FRAME#** as soon as **IRDY#** can be asserted.

6. If not already deasserted, **TRDY#**, **STOP#**, and **DEVSEL#** must be deasserted the clock following the completion of the last data phase and must be tri-stated the next clock.

**Rule 1** means that a data phase can complete with or without **TRDY#** being asserted. When a target is unable to complete a data transfer, it can assert **STOP#** without asserting **TRDY#**.

When both **FRAME#** and **IRDY#** are asserted, the master has committed to complete two data phases. The master is unable to deassert **FRAME#** until the current data phase completes because **IRDY#** is asserted. Because a data phase is allowed to complete when **STOP#** and **IRDY#** are asserted, the master is allowed to start the final data phase by deasserting **FRAME#** and keeping **IRDY#** asserted. The master must deassert **IRDY#** the clock after the completion of the last data phase.

**Rule 2** indicates that data transfers regardless of the state of **STOP#** when both **TRDY#** and **IRDY#** are asserted.

**Rule 3** means that once **STOP#** is asserted, it must remain asserted until the transaction is complete. The last data phase of a transaction completes when **FRAME#** is deasserted, **IRDY#** is asserted, and **STOP#** (or **TRDY#**) is asserted. The target must not assume any timing relationship between the assertion of **STOP#** and the deassertion of **FRAME#**, but must keep **STOP#** asserted until **FRAME#** is deasserted and **IRDY#** is asserted (the last data phase completes). **STOP#** must be deasserted on the clock following the completion of the last data phase.

When both **STOP#** and **TRDY#** are asserted in the same data phase, the target will transfer data in that data phase. In this case, **TRDY#** must be deasserted when the data phase completes. As before, **STOP#** must remain asserted until the transaction ends whereupon it is deasserted.

If the target requires wait states in the data phase where it asserts **STOP#**, it must delay the assertion of **STOP#** until it is ready to complete the data phase.

**Rule 4** means the target is not allowed to change its mind once it has committed to complete the current data phase. Committing to complete a data phase occurs when the target asserts either **TRDY#** or **STOP#**. The target commits to:

- Transfer data in the current data phase and continue the transaction (if a burst) by asserting **TRDY#** and not asserting **STOP#**
- Transfer data in the current data phase and terminate the transaction by asserting both **TRDY#** and **STOP#**
- Not transfer data in the current data phase and terminate the transaction by asserting **STOP#** and deasserting **TRDY#**
- Not transfer data in the current data phase and terminate the transaction with an error condition (Target-Abort) by asserting **STOP#** and deasserting **TRDY#** and **DEVSEL#**

The target has not committed to complete the current data phase while **TRDY#** and **STOP#** are both deasserted. The target is simply inserting wait states.

**Rule 5** means that when the master samples **STOP#** asserted, it must deassert **FRAME#** on the first cycle thereafter in which **IRDY#** is asserted. The assertion of **IRDY#** and deassertion of **FRAME#** should occur as soon as possible after **STOP#** is asserted, preferably within one to three cycles. This assertion of **IRDY#** (and therefore **FRAME#**

deassertion) may occur as a consequence of the normal **IRDY#** behavior of the master had the current transaction not been target terminated. Alternatively, if **TRDY#** is deasserted (indicating there will be no further data transfer), the master may assert **IRDY#** immediately (even without being prepared to complete a data transfer). If a Memory Write and Invalidate transaction is terminated by the target, the master completes the transaction (the rest of the cacheline) as soon as possible (adhering to the **STOP#** protocol) using the Memory Write command (since the conditions to issue Memory Write and Invalidate are no longer true).

**Rule 6** requires the target to release control of the target signals in the same manner it would if the transaction had completed using master termination. Retry and Disconnect are normal termination conditions on the bus. Only Target-Abort is an abnormal termination that may have caused an error. Because the reporting of errors is optional, the bus must continue operating as though the error never occurred.

### Examples of Target Termination

#### Retry

Figure 3-9 shows a transaction being terminated with Retry. The transaction starts with **FRAME#** asserted on clock 2 and **IRDY#** asserted on clock 3. The master requests multiple data phases because both **FRAME#** and **IRDY#** are asserted on clock 3. The target claims the transaction by asserting **DEVSEL#** on clock 4.

The target determines it cannot complete the master's request and also asserts **STOP#** on clock 4 while keeping **TRDY#** deasserted. The first data phase completes on clock 4 because both **IRDY#** and **STOP#** are asserted. Since **TRDY#** was deasserted, no data was transferred during the initial data phase. Because **STOP#** was asserted and **TRDY#** was deasserted on clock 4, the master knows the target is unwilling to transfer any data for this transaction at the present time. The master is required to deassert **FRAME#** as soon as **IRDY#** can be asserted. In this case, **FRAME#** is deasserted on clock 5 because **IRDY#** is asserted on clock 5. The last data phase completes on clock 5 because **FRAME#** is deasserted and **STOP#** is asserted. The target deasserts **STOP#** and **DEVSEL#** on clock 6 because the transaction is complete. This transaction consisted of two data phases in which no data was transferred and the master is required to repeat the request again.

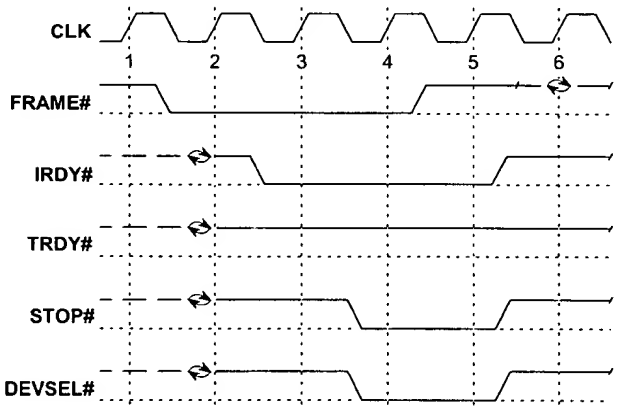



**Figure 3-9: Retry**

### Disconnect With Data

Disconnect - A, in Figure 3-10, is where the master is inserting a wait state when the target signals Disconnect with data. This transaction starts prior to clock 1. The current data phase, which could be the initial or a subsequent data phase, completes on clock 3. The master inserts a wait state on clocks 1 and 2, while the target inserts a wait state only on clock 1. Since the target wants to complete only the current data phase, and no more, it asserts **TRDY#** and **STOP#** at the same time. In this example, the data is transferred during the last data phase. Because the master sampled **STOP#** asserted on clock 2, **FRAME#** is deasserted on clock 3 and the master is ready to complete the data phase (**IRDY#** is asserted). Since **FRAME#** is deasserted on clock 3, the last data phase completes because **STOP#** is asserted and data transfers because both **IRDY#** and **TRDY#** are asserted. Notice that **STOP#** remains asserted for both clocks 2 and 3. The target is required to keep **STOP#** asserted until **FRAME#** is deasserted.

Disconnect - B, in Figure 3-10, is almost the same as Disconnect - A, but **TRDY#** is not asserted in the last data phase. In this example, data was transferred on clocks 1 and 2 but not during the last data phase. The target indicates that it cannot continue the burst by asserting both **STOP#** and **TRDY#** together. When the data phase completes on clock 2, the target is required to deassert **TRDY#** and keep **STOP#** asserted. The last data phase completes, without transferring data, on clock 3 because **TRDY#** is deasserted and **STOP#** is asserted. In this example, there are three data phases, two that transfer data and one that does not.

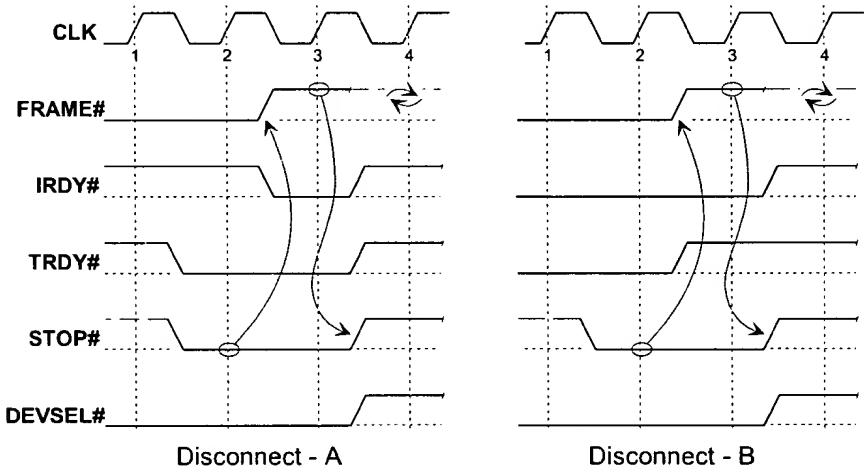



**Figure 3-10: Disconnect With Data**

Figure 3-11 is an example of Master Completion termination where the target blindly asserts **STOP#**. This is a legal termination where the master is requesting a transaction with a single data phase and the target blindly asserts **STOP#** and **TRDY#** indicating it can complete only a single data phase. The transaction starts like all transactions with the assertion of **FRAME#**. The master indicates that the initial data phase is the final data phase because **FRAME#** is deasserted and **IRDY#** is asserted on clock 3. The target claims the transaction, indicates it is ready to transfer data, and requests the transaction to stop by asserting **DEVSEL#**, **TRDY#**, and **STOP#** all at the same time.

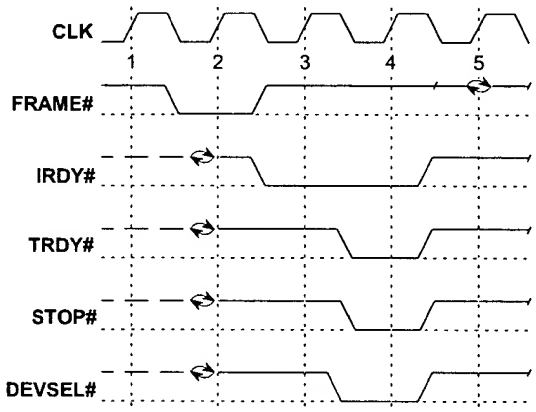



**Figure 3-11: Master Completion Termination**

### Disconnect Without Data

Figure 3-12 shows a transaction being terminated with Disconnect without data. The transaction starts with **FRAME#** being asserted on clock 2 and **IRDY#** being asserted on clock 3. The master is requesting multiple data phases because both **FRAME#** and **IRDY#** are asserted on clock 3. The target claims the transaction by asserting **DEVSEL#** on clock 4.

The first data phase completes on clock 4 and the second completes on clock 5. On clock 6, the master wants to continue bursting because **FRAME#** and **IRDY#** are still asserted. However, the target cannot complete any more data phases and asserts **STOP#** and deasserts **TRDY#** on clock 6. Since **IRDY#** and **STOP#** are asserted on clock 6, the third data phase completes. The target continues to keep **STOP#** asserted on clock 7 because **FRAME#** is still asserted on clock 6. The fourth and final data phase completes on clock 7 since **FRAME#** is deasserted (**IRDY#** is asserted) and **STOP#** is asserted on clock 7. The bus returns to the Idle state on clock 8.

In this example, the first two data phases complete transferring data while the last two do not. This might happen if a device accepted two DWORDs of data and then determined that its buffers were full or if the burst crossed a resource boundary. The target is able to complete the first two data phases but cannot complete the third. When and if the master continues the burst, the device that owns the address of the next untransferred data will claim the access and continue the burst.

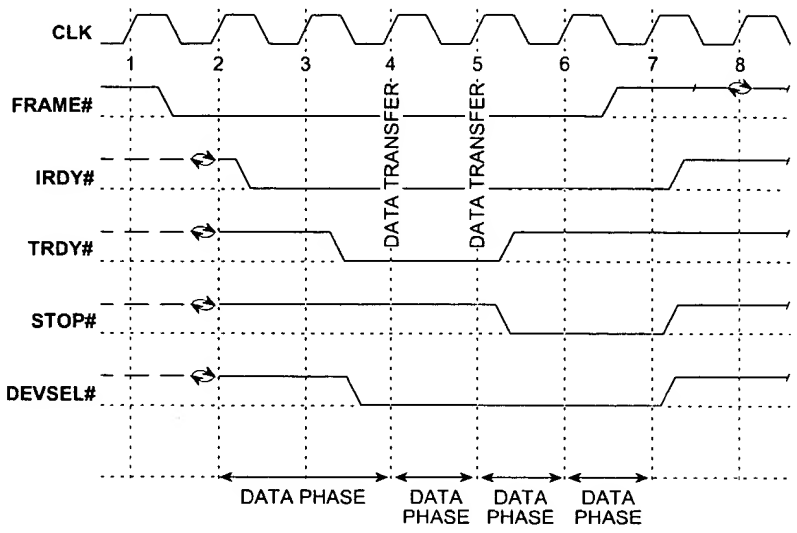



**Figure 3-12: Disconnect-1 Without Data Termination**

Figure 3-13 shows the same transaction as described in Figure 3-12 except that the master inserts a wait state on clock 6. Since **FRAME#** was not deasserted on clock 5, the master committed to at least one more data phase and must complete it. The master is not allowed simply to transition the bus to the Idle state by deasserting **FRAME#** and keeping **IRDY#** deasserted. This would be a violation of bus protocol. When the master is ready to assert **IRDY#**, it deasserts **FRAME#** indicating the last data phase, which completes on clock 7 since **STOP#** is asserted. This example only consists of three data phases while the previous had four. The fact that the master inserted a wait state allowed the master to complete the transaction with the third data phase. However, from a clock count, the two transactions are the same.

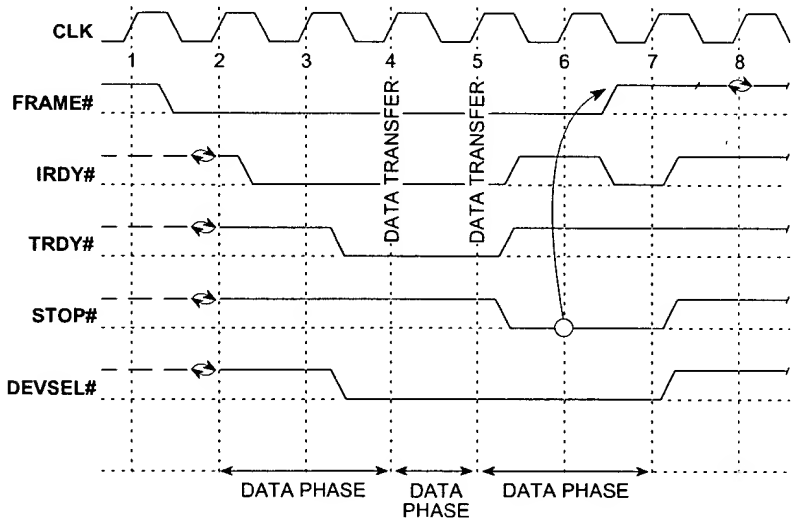



**Figure 3-13: Disconnect-2 Without Data Termination**

### Target-Abort

Figure 3-14 shows a transaction being terminated with Target-Abort. Target-Abort indicates the target requires the transaction to be stopped and does not want the master to repeat the request again. Sometime prior to clock 1, the master asserted **FRAME#** to initiate the request and the target claimed the access by asserting **DEVSEL#**. Data phases may or may not have completed prior to clock 1. The target determines that the master has requested a transaction that the target is incapable of completing or has determined that a fatal error has occurred. Before the target can signal Target-Abort, **DEVSEL#** must be asserted for one or more clocks. To signal Target-Abort, **TRDY#** must be deasserted when **DEVSEL#** is deasserted and **STOP#** is asserted, which occurs on clock 2. If any data was transferred during the previous data phases of the current transaction, it may have been corrupted. Because **STOP#** is asserted on clock 2 and the master can assert **IRDY#** on clock 3, the master deasserts **FRAME#** on clock 3. The transaction completes on clock 3 because **IRDY#** and **STOP#** are asserted. The master deasserts **IRDY#** and the target deasserts **STOP#** on clock 4.

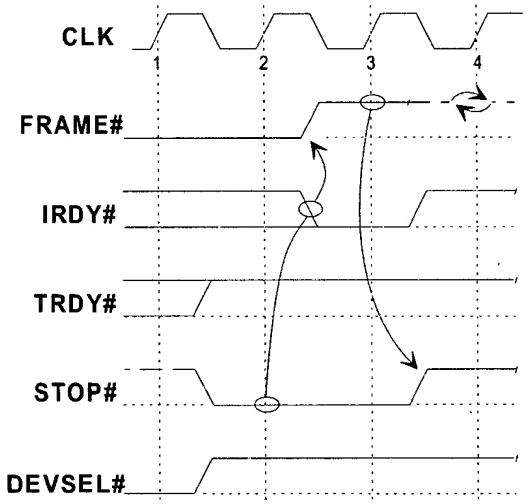



**Figure 3-14: Target-Abort**

## 3.3.3.2.2. Requirements on a Master Because of Target Termination

Although not all targets will implement all forms of target termination, masters must be capable of properly dealing with them all.

### Deassertion of REQ# When Target Terminated

When the current transaction is terminated by the target either by Retry or Disconnect (with or without data), the master must deassert its **REQ#** signal before repeating the transaction. A device containing a single source of master activity must deassert **REQ#** for a minimum of two clocks, one being when the bus goes to the Idle state (at the end of the transaction where **STOP#** was asserted) and either the clock before or the clock after the Idle state.

Some devices contain multiple sources of master activity that share the same **REQ#** pin. Examples of such devices include the following:

- A single function device that contains two independent sub-functions. One that produces data and one that consumes data.
- A multi-function device.
- A PCI-to-PCI bridge that is capable of forwarding multiple Delayed Transactions from the other bus.

A device containing multiple sources of master activity that share a single **REQ#** pin is permitted to allow each source to use the bus (assuming that **GNT#** is still asserted) without deasserting **REQ#** even if one or more sources are target terminated (**STOP#** asserted). However, the device must deassert **REQ#** for two consecutive clocks, one of which while the bus is Idle, before any transaction that was target terminated can be repeated.

The master is not required to deassert its **REQ#** when the target requests the transaction to end by asserting **STOP#** in the last data phase. An example is Figure 3-11 which is really Master Completion termination and not target termination.

### Repeat Request Terminated With Retry

A master which is target terminated with Retry must unconditionally repeat the same request until it completes; however, it is not required to repeat the transaction when terminated with Disconnect. "Same request" means that the same address, same command, same byte enables, same write data for write transactions (even if the byte enable for that byte lane is not asserted), and, if supported, **LOCK#** and **REQ64#** that were used on the original request must be used when the access is repeated. "Unconditionally" in the above rule means the master must repeat the same transaction that was terminated with Retry independent of any subsequent events (except as noted below) until the original transaction is satisfied.

This does not mean the master must immediately repeat the same transaction. In the simplest form, the master would request use of the bus after the two clocks **REQ#** was deasserted and repeat the same transaction. The master is permitted to perform other bus transactions, but cannot require them to complete before repeating the original transaction. If the device also implements target functionality, it must be able to accept accesses during this time as well.

A multi-function device is a good example of how this works. Functions 1, 2, and 3 of a single device are all requesting use of the interface. Function 1 requests a read transaction and is terminated with Retry. Once Function 1 has returned the bus to an Idle state, Function 2 may attempt a transaction (assuming **GNT#** is still active for the device). After Function 2 releases the bus, Function 3 may proceed if **GNT#** is still active. Once Function 3 completes, the device must deassert its **REQ#** for the two clocks before reasserting it. As illustrated above, Function 1 is not required to complete its transaction before another function can request a transaction. But Function 1 must repeat its access regardless of how the transactions initiated by Function 2 or 3 are terminated. The master of a transaction must repeat its transaction unconditionally, which means the repeat of the transaction cannot be gated by any other event or condition.

This rule applies to all transactions that are terminated by Retry regardless of how many previous transactions may have been terminated by Retry. In the example above, if Function 2 attempted to do a transaction and was terminated by Retry, it must repeat that

transaction unconditionally just as Function 1 is required to repeat its transaction unconditionally. Neither Function 1 nor Function 2 can depend on the completion of the other function's transaction or the success of any transaction attempted by Function 3 to be able to repeat its original request.

A subsequent transaction (not the original request) could result in the assertion of **SERR#**, **PERR#**, or being terminated with Retry, Disconnect, Target-Abort, or Master-Abort. Any of these events would have no effect on the requirement that the master must repeat an access that was terminated with Retry.

A master should repeat a transaction terminated by Retry as soon as possible, preferably within 33 clocks. However, there are a few conditions when a master is unable to repeat the request. These conditions typically are caused when an error occurs; for example, the system asserts **RST#**, the device driver resets, and then re-initializes the component, or software disables the master by resetting the Bus Master bit (bit 2 in the Command register). Refer to Section 3.3.3.3.3. for a description of how a target using Delayed Transaction termination handles this error condition.

However, when the master repeats the transaction and finally is successful in transferring data, it is not required to continue the transaction past the first data phase.

**Implementation Note: Potential Temporary Deadlock and Resulting Performance Impacts**

The previous paragraph states that a master may perform other bus transactions, but cannot require them to complete before repeating the original transaction (one previously target terminated with Retry). If a master does not meet this requirement, it may cause temporary deadlocks resulting in significant device and system performance impacts. Devices designed prior to Revision 2.1 of this specification may exhibit this behavior. Such temporary deadlocks should eventually clear when the discard timer (refer to Section 3.3.3.3.3.) expires.

## 3.3.3.3. Delayed Transactions

Delayed Transaction termination is used by targets that cannot complete the initial data phase within the requirements of this specification. There are two types of devices that will use Delayed Transactions: I/O controllers and bridges (in particular, PCI-to-PCI bridges). In general, I/O controllers will handle only a single Delayed Transaction at a time, while bridges may choose to handle multiple transactions to improve system performance.

One advantage of a Delayed Transaction is that the bus is not held in wait states while completing an access to a slow device. While the originating master rearbitrates for the bus, other bus masters are allowed to use the bus bandwidth that would normally be wasted holding the master in wait states. Another advantage is that all posted (memory write) data is not required to be flushed before the request is accepted. The actual flushing of the posted memory write data occurs before the Delayed Transaction completes on the originating bus. This allows posting to remain enabled while a non-postable transaction completes and still maintains the system ordering rules.

The following discussion focuses on the basic operation and requirements of a device that supports a single Delayed Transaction at a time. Section 3.3.3.3.5. extends the basic concepts from support of a single Delayed Transaction to the support of multiple Delayed Transactions at a time.

### 3.3.3.3.1. Basic Operation of a Delayed Transaction

All bus commands that must complete on the destination bus before completing on the originating bus may be completed as a Delayed Transaction. These include Interrupt Acknowledge, I/O Read, I/O Write, Configuration Read, Configuration Write, Memory Read, Memory Read Line, and Memory Read Multiple commands. Memory Write and Memory Write and Invalidate commands can complete on the originating bus before completing on the destination bus (i.e., can be posted). Each command is not completed using Delayed Transaction termination and are either posted or terminated with Retry. For I/O controllers, the term *destination bus* refers to the internal bus where the resource addressed by the transaction resides. For a bridge, the destination bus means the interface that was not acting as the target of the original request. For example, the secondary bus of a bridge is the destination bus when a transaction originates on the primary bus of the bridge and targets (addresses) a device attached to the secondary bus of the bridge. However, a transaction that is moving in the opposite direction would have the primary bus as the destination bus.

A Delayed Transaction progresses to completion in three steps:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first step, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. The master of a request that is terminated with Retry cannot distinguish between a target which is completing the transaction using Delayed Transaction termination and a target which simply cannot complete the transaction at the current time. Since the master cannot tell the difference, it must reissue any request that has been terminated with Retry until the request completes (refer to Section 3.3.3.2.2.).

During the second step, the target independently completes the request on the destination bus using the latched information from the Delayed Request. If the Delayed Request is a read, the target obtains the requested data and completion status. If the Delayed Request is a write, the target delivers the write data and obtains the completion status. The result of completing the Delayed Request on the destination bus produces a Delayed Completion, which consists of the latched information of the Delay Request and the completion status (and data if a read request). The target stores the Delayed Completion until the master repeats the initial request.

During the third step, the master successfully rearbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed. The status returned to the master is exactly the same as the target obtained when it executed (completed) the Delayed Request (i.e., Master-Abort, Target-Abort, parity error, normal, Disconnect, etc.).

### 3.3.3.3.2. Information Required to Complete a Delayed Transaction

To complete a transaction using Delayed Transaction termination, a target must latch the following information:

- address
- command
- byte enables
- address and data parity, if the Parity Error Response bit (bit 6 of the command register) is set
- **REQ64#** (if a 64-bit transfer)

For write transactions completed using Delayed Transaction termination, a target must also latch data from byte lanes for which the byte enable is asserted and may optionally latch data from byte lanes for which the byte enable is deasserted. Refer to Appendix F for requirements for a bridge to latch **LOCK#** when completing a Delayed Transaction.

On a read transaction, the address and command are available during the address phase and the byte enables during the following clock. Byte enables for both read and write transactions are valid the entire data phase and are independent of **IRDY#**. On a write transaction, all information is valid at the same time as a read transaction, except for the actual data, which is valid only when **IRDY#** is asserted.

Note: Write data is only valid when **IRDY#** is asserted. Byte enables are always valid for the entire data phase regardless of the state of **IRDY#**.

The target differentiates between transactions (by the same or different masters) by comparing the current transaction with information latched previously (for both Delayed Request(s) and Delayed Completion(s)). During a read transaction, the target is not required to use byte enables as part of the comparison, if all bytes are returned independent of the asserted byte enables and the accessed location has no read side-effects (pre-fetchable). If the compare matches a Delayed Request (already enqueued), the target does not enqueue the request again but simply terminates the transaction with Retry indicating that the target is not yet ready to complete the request. If the compare matches a Delayed Completion, the target responds by signaling the status and providing the data if a read transaction.

The master must repeat the transaction exactly as the original request, including write data in all byte lanes (whether the corresponding byte enables are asserted or not). Otherwise, the target will assume it is a new transaction. If the original transaction is never repeated, it will eventually be discarded when the Discard Timer expires (refer to Section 3.3.3.3.3.). Two masters could request the exact same transaction and the target cannot and need not distinguish between them and will simply complete the access.

Special requirements apply if a data parity error occurs while initiating or completing a Delayed Transaction. Refer to Section 3.7.5. for details about a parity error and Delayed Transactions.

### 3.3.3.3.3. Discarding a Delayed Transaction

A device is allowed to discard a Delayed Request from the time it is enqueued until it has been attempted on the destination bus, since the master is required to repeat the request until it completes. Once a Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus and cannot be discarded. The master is allowed to present other requests. But if it attempts more than one request, the master must continue to repeat all requests that have been attempted unconditionally until they complete. The repeating of the requests is not required to be equal, but is required to be fair.

When a Delayed Request completes on the destination bus, it becomes a Delayed Completion. The target device is allowed to discard Delayed Completions in only two cases. The first case is when the Delayed Completion is a read to a pre-fetchable region (or the command was Memory Read Line or Memory Read Multiple). The second case is for all Delayed Completions (read or write, pre-fetchable or not) when the master has not repeated the request within $2^{15}$ clocks. When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

If the Discard Timer expires, the device may choose to report an error or not. If the data is prefetchable (case 1), it is recommended that the device not report an error, since system integrity is not effected. However, if the data on a read access is not prefetchable (case 2), it is recommended that the device report the error to its device driver since system integrity is affected.

### 3.3.3.3.4. Memory Writes and Delayed Transactions

While completing a Delayed Request, the target is also required to complete all memory write transactions addressed to it. The target may, from time to time, retry a memory write while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full, or before the Delayed Request has completed on the destination bus (but is guaranteed to complete). However, the target cannot require the Delayed Transaction to complete on the originating bus before accepting the memory write data; otherwise, a deadlock may occur. Refer to Section 3.10., item 6, for additional information. The following implementation note describes the deadlock.

**Implementation Note: Deadlock When Memory Write Data is Not Accepted**

The deadlock occurs when the master and the target of a transaction reside on different buses (or segments). The PCI-to-PCI bridge[18] that connects the two buses together does not implement Delayed Transactions. The master initiates a request that is forwarded to the target by the bridge. The target responds to the request by using Delayed Transaction termination (terminated with Retry). The bridge terminates the master's request with Retry (without latching the request). Another master (on the same bus segment as the original master) posts write data into the bridge targeted at the same device as the read request. Because it is designed to the previous version of this specification, before Delayed Transactions, the bridge is required to flush the memory write data before the read can be repeated. If the target that uses Delayed Transaction termination will not accept the memory write data until the master repeats the initial read, a deadlock occurs because the bridge cannot repeat the request until the target accepts the write data. To prevent this from occurring, the target that uses the Delayed Transaction termination to meet the initial latency requirements is required to accept memory write data even though the Delayed Transaction has not completed.

## 3.3.3.3.5. Supporting Multiple Delayed Transactions

This section takes the basic concepts of a single Delayed Transaction as described in the previous section and extends them to support multiple Delayed Transactions at the same time. Bridges (in particular, PCI-to-PCI bridges) are the most likely candidates to handle multiple Delayed Transactions as a way to improve system performance and meet the initial latency requirements. To assist in understanding the requirements of supporting multiple Delayed Transactions, the following section focuses on a PCI-to-PCI bridge. This focus allows the same terminology to be used when describing transactions initiated on either interface of the bridge. Most other bridges (host bus bridge and standard expansion bus bridge) will typically handle only a single Delayed Transaction. Supporting multiple transactions is possible but the details may vary. The fundamental requirements in all cases are that transaction ordering be maintained as described in Section 3.2.5. and Section 3.3.3.3.4. and deadlocks will be avoided.

**Transaction Definitions**

**PMW** - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

**DRR** - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an Interrupt Acknowledge, I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple command. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Until that time, the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

**DWR** - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write

[18] This is a bridge that is built to an earlier version of this specification.

command. Note: Memory Write and Memory Write and Invalidate commands must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Until that time, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

**DRC** - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the status of the target (normal, Master-Abort, Target-Abort, parity error, etc.).

**DWC** - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access but only status of how it completed (normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

### Ordering Rules for Multiple Delayed Transactions

Table 3-3 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. The number of simultaneous transactions is limited by the implementation and not by the architecture. Because there are five types of transactions that can be handled in each direction, the following table has 25 entries. Of the 25 boxes in the table, only four are required No's, eight are required Yes's, and the remaining 13 are don't cares. The column of the table represents an access that was accepted previously by the bridge, while the row represents a transaction that was accepted subsequent to the access represented by the column. The following table specifies the ordering relationships between transactions as they cross a bridge. For an explanation as to why these rules are required or for a general discussion on system ordering rules, refer to Appendix E for details.

**Table 3-3: Ordering Rules for Multiple Delayed Transactions**

| Row pass Col.? | PMW (Col 2) | DRR (Col 3) | DWR (Col 4) | DRC (Col 5) | DWC (Col 6) |
|---|---|---|---|---|---|
| PMW (Row 1) | No | Yes | Yes | Yes | Yes |
| DRR (Row 2) | No | Yes/No | Yes/No | Yes/No | Yes/No |
| DWR (Row 3) | No | Yes/No | Yes/No | Yes/No | Yes/No |
| DRC (Row 4) | No | Yes | Yes | Yes/No | Yes/No |
| DWC (Row 5) | Yes/No | Yes | Yes | Yes/No | Yes/No |

**No** - indicates the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes are found in column 2 and maintain a consistent view of data in the system as described by the Producer - Consumer Model found in Appendix E. These boxes prevent PMW data from being passed by other accesses.

**Yes** - The four Yes boxes in Row 1 indicate the PMW must be allowed to complete before Delayed Requests or Delayed Completions moving in the same direction or a deadlock can occur. This prevents deadlocks from occurring when Delayed Transactions are used with devices designed to an earlier version of this specification. A PMW cannot be delayed from completing because a Delayed Request or a Delayed Completion was accepted prior to the PMW. The only thing that can prevent the PMW from completing is gaining access to the bus or the target terminating the attempt with Retry. Both conditions are temporary and will resolve independently of other events. If the master

continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. There is no ordering violation when a subsequent transaction completes before a prior transaction.

The four Yes boxes in rows 4 and 5, columns 3 and 4, indicate that Delayed Completions must be allowed to pass Delayed Requests moving in the same direction. This prevents deadlocks from occurring when two bridges that support Delayed Transactions are requesting accesses to each other. If neither bridge allows Delayed Completions to pass the Delayed Requests, neither can make progress.

**Yes/No** - indicates the bridge may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

### Ordering of Delayed Transactions

The ordering of Delayed Transactions is established when the transaction completes on the originating bus (i.e., the requesting master receives a response other than Retry). Delayed Requests and Delayed Completions are intermediate steps in the process of completing a Delayed Transaction, which occur prior to the completion of the transaction on the originating bus. As a result, reordering is allowed for Delayed Requests with respect to other Delayed Requests, Delayed Requests with respect to Delayed Completions, or for Delayed Completions with respect to other Delayed Completions. However, reordering is not allowed with respect to memory write transactions, which is described in Table 3-3 (the No boxes).

In general, a master does not need to wait for one request to be completed before it issues another request. As described in Section 3.3.3.2.2., a master may have any number of requests terminated with Retry at one time, some of which may be serviced as Delayed Transactions and some not. However, if the master does issue a second request before the first is completed, the master must continue to repeat each of the requests fairly, so that each has a fair opportunity to be completed. If a master has a specific need for two transactions to be completed in a particular order, it must wait for the first one to complete before requesting the second.

## 3.4. Arbitration

In order to minimize access latency, the PCI arbitration approach is access-based rather than time-slot-based. That is, a bus master must arbitrate for each access it performs on the bus. PCI uses a central arbitration scheme, where each master agent has a unique request (**REQ#**) and grant (**GNT#**) signal. A simple request-grant handshake is used to gain access to the bus. Arbitration is "hidden," which means it occurs during the previous access so that no PCI bus cycles are consumed due to arbitration, except when the bus is in an Idle state.

An arbitration algorithm must be defined to establish a basis for a worst case latency guarantee. However, since the arbitration algorithm is fundamentally not part of the bus specification, system designers may elect to modify it, but must provide for the latency requirements of their selected I/O controllers and for expansion boards. Refer to Section 3.5.4. for information on latency guidelines. The bus allows back-to-back transactions by the same agent and allows flexibility for the arbiter to prioritize and weight requests. An arbiter can implement any scheme as long as it is fair and only a single **GNT#** is asserted on any rising clock.

The arbiter is required to implement a fairness algorithm to avoid deadlocks. In general, the arbiter must advance to a new agent when the current master deasserts its **REQ#**. Fairness means that each potential master must be granted access to the bus independent of other requests. However, this does not mean that all agents are required to have equal access to the bus. By requiring a fairness algorithm, there are no special conditions to handle when **LOCK#** is active (assuming a resource lock). A system that uses a fairness algorithm is still considered fair if it implements a complete bus lock instead of resource lock. However, the arbiter must advance to a new agent if the initial transaction attempting to establish the lock is terminated with Retry.

## Implementation Note: System Arbitration Algorithm

One example of building an arbiter to implement a fairness algorithm is when there are two levels to which bus masters are assigned. In this example, the agents that are assigned to the first level have a greater need to use the bus than agents assigned to the second level (i.e., lower latency or greater throughput). Second level agents have equal access to the bus with respect to other second level agents. However, the second level agents as a group have equal access to the bus as each agent of the first level. An example of how a system may assign agents to a given level is where devices such as video, ATM, or FDDI bus masters would be assigned to Level 1 while devices such as SCSI, LAN, or standard expansion bus masters would be assigned to the second level.

The figure below is an example of a fairness arbitration algorithm that uses two levels of arbitration. The first level consists of Agent A, Agent B, and Level 2, where Level 2 is the next agent at that level requesting access to the bus. Level 2 consists of Agent X, Agent Y, and Agent Z. If all agents on level 1 and 2 have their **REQ#** lines asserted and continue to assert them, and if Agent A is the next to receive the bus for Level 1 and Agent X is the next for Level 2, then the order of the agents accessing the bus would be:

A, B, Level 2 (this time it is X)

A, B, Level 2 (this time it is Y)

A, B, Level 2 (this time it is Z)

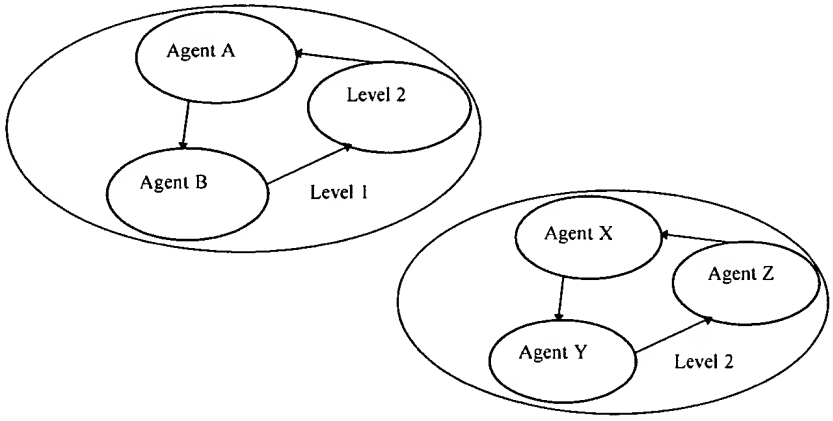
and so forth.

If only Agent B and Agent Y had their **REQ#**s asserted and continued to assert them, the order would be:

B, Level 2 (Y),

B, Level 2 (Y).

By requiring a fairness arbitration algorithm, the system designer can balance the needs of high performance agents such as video, ATM, or FDDI with lower performance bus devices like LAN and SCSI. Another system designer may put only multimedia devices on arbitration Level 1 and put the FDDI (or ATM), LAN, and SCSI devices on Level 2. These examples achieve the highest level of system performance possible for throughput or lowest latency without possible starvation conditions. The performance of the system can be balanced by allocating a specific amount of bus bandwidth to each agent by careful assignment of each master to an arbitration level and programming each agent's Latency Timer appropriately.



## 3.4.1. Arbitration Signaling Protocol

An agent requests the bus by asserting its **REQ#**. Agents must only use **REQ#** to signal a true need to use the bus. An agent must never use **REQ#** to "park" itself on the bus. If bus parking is implemented, it is the arbiter that designates the default owner. When the arbiter determines an agent may use the bus, it asserts the agent's **GNT#**.

The arbiter may deassert an agent's **GNT#** on any clock. An agent must ensure its **GNT#** is asserted on the rising clock edge it wants to start a transaction. Note: A master is allowed to start a transaction when its **GNT#** is asserted and the bus is in an Idle state independent of the state of its **REQ#**. If **GNT#** is deasserted, the transaction must not proceed. Once asserted, **GNT#** may be deasserted according to the following rules:

1. If **GNT#** is deasserted and **FRAME#** is asserted on the same clock, the bus transaction is valid and will continue.
2. One **GNT#** can be deasserted coincident with another **GNT#** being asserted if the bus is not in the Idle state. Otherwise, a one clock delay is required between the deassertion of a **GNT#** and the assertion of the next **GNT#**, or else there may be contention on the **AD** lines and **PAR** due to the current master doing address stepping.
3. While **FRAME#** is deasserted, **GNT#** may be deasserted at any time in order to service a higher priority[19] master or in response to the associated **REQ#** being deasserted.

Figure 3-15 illustrates basic arbitration. Two agents are used to illustrate how an arbiter may alternate bus accesses.

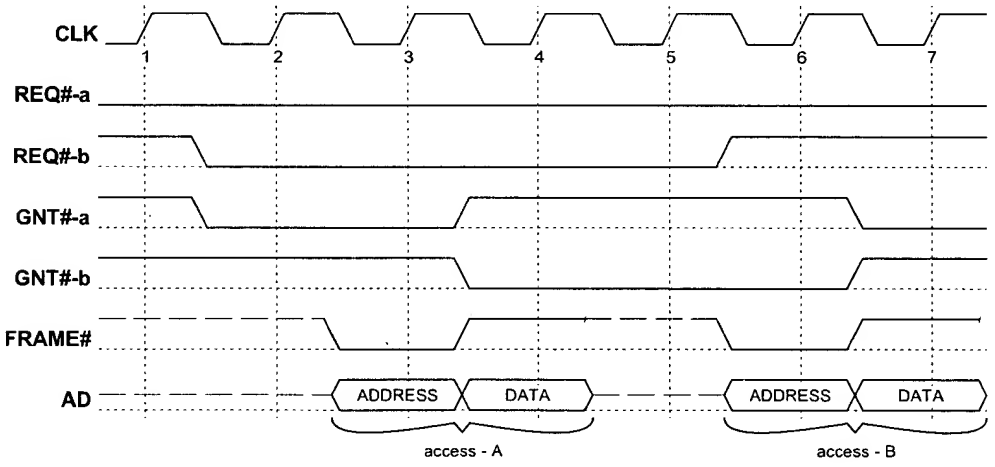



**Figure 3-15: Basic Arbitration**

**REQ#**-a is asserted prior to or at clock 1 to request use of the interface. Agent A is granted access to the bus because **GNT#**-a is asserted at clock 2. Agent A may start a transaction at clock 2 because **FRAME#** and **IRDY#** are deasserted and **GNT#**-a is asserted. Agent A's transaction starts when **FRAME#** is asserted on clock 3. Since

[19] Higher priority here does not imply a fixed priority arbitration, but refers to the agent that would win arbitration at a given instant in time.

Agent A desires to perform another transaction, it leaves **REQ#**-a asserted. When **FRAME#** is asserted on clock 3, the arbiter determines Agent B should go next and asserts **GNT#**-b and deasserts **GNT#**-a on clock 4.

When agent A completes its transaction on clock 4, it relinquishes the bus. All PCI agents can determine the end of the current transaction when both **FRAME#** and **IRDY#** are deasserted. Agent B becomes the owner on clock 5 (because **FRAME#** and **IRDY#** are deasserted) and completes its transaction on clock 7.

Notice that **REQ#**-b is deasserted and **FRAME#** is asserted on clock 6 indicating agent B requires only a single transaction. The arbiter grants the next transaction to Agent A because its **REQ#** is still asserted.

The current owner of the bus keeps **REQ#** asserted when it requires additional transactions. If no other requests are asserted or the current master has highest priority, the arbiter continues to grant the bus to the current master.

### Implementation Note: Bus Parking

When no **REQ#**s are asserted, it is recommended not to remove the current master's **GNT#** to park the bus at a different master until the bus enters its Idle state. If the current bus master's **GNT#** is deasserted, the duration of the current transaction is limited to the value of the Latency Timer. If the master is limited by the Latency Timer, it must rearbitrate for the bus which would waste bus bandwidth. It is recommended to leave **GNT#** asserted at the current master (when no other **REQ#**s are asserted) until the bus enters its Idle state. When the bus is in the Idle state and no **REQ#**s are asserted, the arbiter may park the bus at any agent it desires.

**GNT#** gives an agent access to the bus for a single transaction. If an agent desires another access, it should continue to assert **REQ#**. An agent may deassert **REQ#** anytime, but the arbiter may interpret this to mean the agent no longer requires use of the bus and may deassert its **GNT#**. An agent should deassert **REQ#** in the same clock **FRAME#** is asserted if it only wants to do a single transaction. When a transaction is terminated by a target (**STOP#** asserted), the master must deassert its **REQ#** for a minimum of two clocks, one being when the bus goes to the Idle state (at the end of the transaction where **STOP#** was asserted), and the other being either the clock before or the clock after the Idle state. For an exception, refer to Section 3.3.3.2.1. This allows another agent to use the interface while the previous target prepares for the next access.

The arbiter can assume the current master is "broken" if it has not started an access after its **GNT#** has been asserted (its **REQ#** is also asserted) and the bus is in the Idle state for 16 clocks. The arbiter is allowed to ignore any "broken" master's **REQ#** and may optionally report this condition to the system. However, the arbiter may remove **GNT#** at any time to service a higher priority agent. A master that has requested use of the bus that does not assert **FRAME#** when the bus is in the Idle state and its **GNT#** is asserted faces the possibility of losing its turn on the bus. Note: In a busy system, a master that delays the assertion of **FRAME#** runs the risk of starvation because the arbiter may grant the bus to another agent. For a master to ensure that it gains access to the bus, it must assert **FRAME#** the first clock possible when **FRAME#** and **IRDY#** are deasserted and its **GNT#** is asserted.

## 3.4.2. Fast Back-to-Back Transactions

There are two types of fast back-to-back transactions that can be initiated by the same master: those that access the same agent and those that do not. Fast back-to-back transactions are allowed on PCI when contention on **TRDY#**, **DEVSEL#**, **STOP#**, or **PERR#** is avoided.

The first type of fast back-to-back support places the burden of avoiding contention on the master, while the second places the burden on all potential targets. The master may remove the Idle state between transactions when it can guarantee that no contention occurs. This can be accomplished when the master's current transaction is to the same target as the previous write transaction. This type of fast back-to-back transaction requires the master to understand the address boundaries of the potential target; otherwise, contention may occur. This type of fast back-to-back is optional for a master but must be decoded by a target. The target must be able to detect a new assertion of **FRAME#** (from the same master) without the bus going to the Idle state.

The second type of fast back-to-back support places the burden of no contention on all potential targets. The Fast Back-to-Back Capable bit in the Status register may be hardwired to a logical one (high) if, and, only if, the device, while acting as a bus target, meets the following two requirements:

1. The target must not miss the beginning of a bus transaction, nor lose the address, when that transaction is started without a bus Idle state preceding the transaction. In other words, the target is capable of following a bus state transition from a final data transfer (**FRAME#** high, **IRDY#** low) directly to an address phase (**FRAME#** low, **IRDY#** high) on consecutive clock cycles. Note: The target may or may not be selected on either or both of these transactions, but must track bus states nonetheless.[20]

2. The target must avoid signal conflicts on **DEVSEL#**, **TRDY#**, **STOP#**, and **PERR#**. If the target does not implement the fastest possible **DEVSEL#** assertion time, this guarantee is already provided. For those targets that do perform zero wait state decodes, the target must delay assertion of these four signals for a single clock, except in either one of the following two conditions:

    a. The current bus transaction was immediately preceded by a bus Idle state; that is, this is not a back-to-back transaction, or,

    b. The current target had driven **DEVSEL#** on the previous bus transaction; that is, this is a back-to-back transaction involving the same target as the previous transaction.

    Note: Delaying the assertion of **DEVSEL#** to avoid contention on fast back-to-back transactions does not affect the decode speed indicated in the status register. A device that normally asserts fast **DEVSEL#** still indicates "fast" in the status register even though **DEVSEL#** is delayed by one clock in this case. The status bits associated with decode time are used by the system to allow the subtractive decoding agent to move in the time when it claims unclaimed accesses. However, if the

[20] It is recommended that this be done by returning the target state machine (refer to Appendix B) from the B_BUSY state to the IDLE state as soon as **FRAME#** is deasserted and the device's decode time has been met (a miss occurs) or when **DEVSEL#** is asserted by another target and not waiting for a bus Idle state (**IRDY#** deasserted).

subtractive decode agent claims the access during medium or slow decode time instead of waiting for the subtractive decode time, it must delay the assertion of **DEVSEL#** when a fast back-to-back transaction is in progress; otherwise, contention on **DEVSEL#**, **STOP#**, **TRDY#**, and **PERR#** may occur.

For masters that want to perform fast back-to-back transactions that are supported by the target mechanism, the Fast Back-to-Back Enable bit in the Command register is required. (This bit is only meaningful in devices that act as bus masters and is fully optional.) It is a read/write bit when implemented. When set to a one (high), the bus master may start a PCI transaction using fast back-to-back timing without regard to which target is being addressed providing the previous transaction was a write transaction issued by the current bus master. If this bit is set to a zero (low) or not implemented, the master may perform fast back-to-back only if it can guarantee that the new transaction goes to the same target as the previous one (master based mechanism).

This bit would be set by the system configuration routine after ensuring that all targets on the same bus had the Fast Back-to-Back Capable Bit set.

Note: The master based fast back-to-back mechanism does not allow these fast cycles to occur with separate targets while the target based mechanism does.

If the target is unable to provide both of the guarantees specified above, it must not implement this bit at all, and it will automatically be returned as a zero when the Status register is read.

Fast back-to-back transactions allow agents to utilize bus bandwidth more effectively. It is recommended that targets and those masters that can improve bus utilization should implement this feature, particularly since the implementation cost is negligible.

Under all other conditions, the master must insert a minimum of one Idle bus state. (Also there is always at least one Idle bus state between transactions by different masters.) Note: The master is required to cause an Idle state to appear on the bus when the requirements for a fast back-to-back transaction are not met or when bus ownership changes.

During a fast back-to-back transaction, the master starts the next transaction immediately without an Idle bus state (assuming its **GNT#** is still asserted). If **GNT#** is deasserted in the last data phase of a transaction, the master has lost access to the bus and must relinquish the bus to the next master. The last data phase completes when **FRAME#** is deasserted, and **IRDY#** and **TRDY#** (or **STOP#**) are asserted. The current master starts another transaction on the clock following the completion of the last data phase of the previous transaction.

It is important to note that agents not involved in a fast back-to-back transaction sequence cannot (and generally need not) distinguish intermediate transaction boundaries using only **FRAME#** and **IRDY#** (there is no bus Idle state). During fast back-to-backs only, the master and target involved need to distinguish these boundaries. When the last transaction is over, all agents will see an Idle state. However, those that do support the target based mechanism must be able to distinguish the completion of all PCI transactions and be able to detect all address phases.

In Figure 3-16, the master completes a write on clock 3 and starts the next transaction on clock 4. The target must begin sampling **FRAME#** on clock 4 since the previous transaction completed on clock 3; otherwise, it will miss the address of the next transaction. A device must be able to decode back-to-back operations, to determine if it

is the current target, while a master may optionally support this function. A target is free to claim ownership by asserting **DEVSEL#**, then Retry the request.

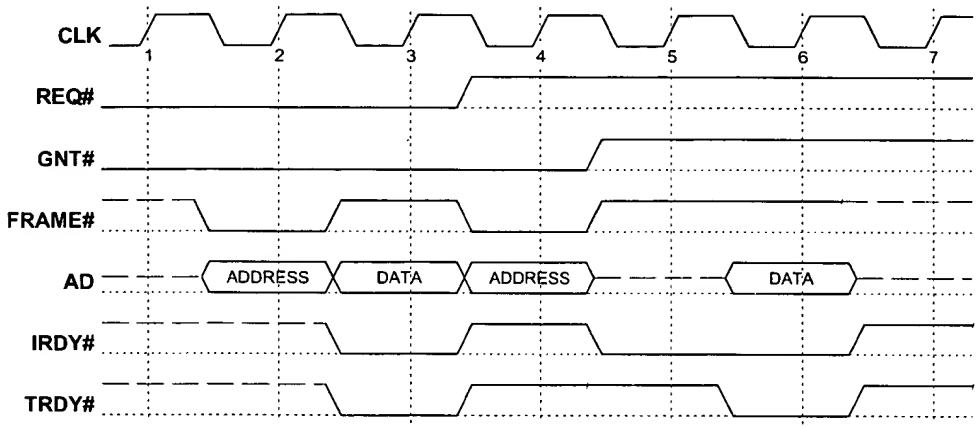



**Figure 3-16: Arbitration for Back-to-Back Access**

## 3.4.3. Arbitration Parking

The term *park* implies permission for the arbiter to assert **GNT#** to a selected agent when no agent is currently using or requesting the bus. The arbiter can select the default owner any way it wants (fixed, last used, etc.) or can choose not to park at all (effectively designating itself the default owner). When the arbiter asserts an agent's **GNT#** and the bus is in the Idle state, that agent must enable its **AD[31::00]**, **C/BE[3::0]#**, and (one clock later) **PAR** output buffers within eight clocks (required), while two-three clocks is recommended. (Refer to Section 3.7.1. for a description of the timing relationship of **PAR** to **AD**). The agent is not compelled to turn on all buffers in a single clock. This requirement ensures that the arbiter can safely park the bus at some agent and know that the bus will not float. (If the arbiter does not park the bus, the central resource device in which the arbiter is embedded typically drives the bus.)

If the bus is in the Idle state and the arbiter removes an agent's **GNT#**, the agent has lost access to the bus except for one case. The one case is if the arbiter deasserted **GNT#** coincident with the agent asserting **FRAME#**. In this case, the master will continue the transaction. Otherwise, the agent must tri-state **AD[31::00]**, **C/BE#[3::0]**, and (one clock later) **PAR**. Unlike above, the agent must disable all buffers in a single clock to avoid possible contention with the next bus owner.

Given the above, the minimum arbitration latency achievable on PCI from the bus Idle state is as follows:

- Parked: zero clocks for parked agent, two clocks for others
- Not Parked: one clock for every agent

When the bus is parked at an agent, the agent is allowed to start a transaction without **REQ#** being asserted. (A master can start a transaction when the bus is in the Idle state and **GNT#** is asserted.) When the agent needs to do multiple transactions, it should assert **REQ#** to inform the arbiter that it intends to do multiple transactions. When a master requires only a single transaction, it should not assert **REQ#**; otherwise, the arbiter may continue to assert its **GNT#** when it does not require use of the bus.

## 3.5. Latency

PCI is a low latency, high throughput I/O bus. Both targets and masters are limited as to the number of wait states they can add to a transaction. Furthermore, each master has a programmable timer limiting its maximum tenure on the bus during times of heavy bus traffic. Given these two limits and the bus arbitration order, worst-case bus acquisition latencies can be predicted with relatively high precision for any PCI bus master. Even bridges to standard expansion buses with long access times (ISA, EISA, or MC) can be designed to have minimal impact on the PCI bus and still keep PCI bus acquisition latency predictable.

### 3.5.1. Target Latency

Target latency is the number of clocks the target waits before asserting **TRDY#**. Requirements on the initial data phase are different from those of subsequent data phases.

#### 3.5.1.1. Target Initial Latency

Target initial latency is the number of clocks from the assertion of **FRAME#** to the assertion of **TRDY#** which completes the initial data phase, or to the assertion of **STOP#** in the Retry and Target-Abort cases. This number of clocks varies depending on whether the command is a read or write, and, if a write, whether it can be posted or not. A memory write command should simply be posted by the target in a buffer and written to the final destination later. In this case, the target initial latency is small because the transaction was simply a register to register transfer. Meeting target initial latency on read transactions is more difficult since this latency is a combination of the access time of the storage media (e.g., disk, DRAM, etc.) and the delay of the interface logic. Meeting initial latency on I/O and configuration write transactions are similar to read latency.

Target initial latency requirements depend on the state of system operation. The system can either be operating in initialization-time or run-time. Initialization-time begins when **RST#** is deasserted and completes $2^{25}$ PCI clocks later. Run-time follows initialization-time.

If a target is accessed during initialization-time, it is allowed to do any of the following:

- Ignore the request (except if it is a boot device)
- Claim the access and hold in wait states until it can complete the request, not to exceed the end of initialization-time
- Claim the access and terminate with Retry

If a target is accessed during run-time (**RST#** has been deasserted greater than $2^{25}$ clocks), it must complete the initial data phase of a transaction (read or write) within 16 clocks from the assertion of **FRAME#**. The target completes the initial data phase by asserting **TRDY#** (to accept or provide the requested data) or by terminating the request by asserting **STOP#** within the target initial latency requirement.

Host bus bridges are granted an additional 16 clocks, to a maximum of 32 clocks, to complete the initial data phase when the access hits a modified line in a cache. However, the host bus bridge can never exceed 32 clocks on any initial data phase.

In most designs, the initial data phase latency is known when the device is designed. If the time required to complete the initial data phase will normally exceed the maximum target initial latency specification, the device must terminate the transaction with Retry as soon as possible and execute the transaction as a Delayed Transaction.

In the unusual case in which the initial data phase latency cannot be determined in advance, the target is allowed to implement a counter that causes the target to assert **STOP#** and to begin execution of the transaction as a Delayed Transaction on or before the sixteenth clock, if **TRDY#** is not asserted sooner. A target device that waits for an initial data phase latency counter to expire prior to beginning a Delayed Transaction reduces PCI bandwidth available to other agents and limits transaction efficiency. Therefore, this behavior is strongly discouraged.

### Implementation Note: Working with Older Targets that Violate the Target Initial Latency Specification

All new target devices must adhere to the 16 clock initial latency requirement except as noted above. However, a new master should not depend on targets meeting the 16 clock maximum initial access latency for functional operation (in the near term), but must function normally (albeit with reduced performance) since systems and devices were designed and built against an earlier version of this specification and may not meet the new requirements. New devices should work with existing devices.

Three options are given to targets to meet the initial latency requirements. Most targets will use either Option 1 or Option 2. Those devices unable to use Option 1 or Option 2 are required to use Option 3.

**Option 1** is for a device that always transfers data (asserts **TRDY#**) within 16 clocks from the assertion of **FRAME#**.

Note: The majority of I/O controllers built prior to revision 2.1 of this specification will meet the initial latency requirements using Option 1. In this case, the target always asserts **TRDY#** to complete the initial data phase of the transaction within 16 clocks of the assertion of **FRAME#**.

**Option 2** is for devices that normally transfer data within 16 clocks, but under some specific conditions will exceed the initial latency requirement. Under these conditions, the device terminates the access with Retry within 16 clocks from the assertion of **FRAME#**.

For devices that cannot use Option 1, a small modification may be required to meet the initial latency requirements as described by Option 2. This option is used by a target that can normally complete the initial data phase within 16 clocks (same as Option 1), but occasionally will take longer and uses the assertion of **STOP#** to meet the initial latency requirement. It then becomes the responsibility of the master to attempt the transaction again at a later time. A target is permitted to do this only when there is a high probability the target will be able to complete the transaction when the master repeats the request; otherwise, the target must use Option 3.

**Implementation Note: An Example of Option 2**

Consider a simple graphic device that normally responds to a request within 16 clocks but under special conditions, such as refreshing the screen, the internal bus is "busy" and prevents data from transferring. In this case, the target terminates the access with Retry knowing the master will repeat the transaction and the target will most likely be able to complete the transfer then.

The device could have an internal signal that indicates to the bus interface unit that the internal bus is busy and data cannot be transferred at this time. This allows the device to claim the access (asserts **DEVSEL#**) and immediately terminate the access with Retry. By doing this instead of terminating the transaction 16 clocks after the assertion of **FRAME#**, other agents can use the bus.

**Option 3** is for a device that frequently cannot transfer data within 16 clocks. This option requires the device to use Delayed Transactions which are discussed in detail in Section 3.3.3.3.

Those devices that cannot meet the requirements of Option 1 or 2 are required to use Option 3. This option is used by devices that under normal conditions cannot complete the transaction within the initial latency requirements. An example could be an I/O controller that has several internal functions contending with the PCI interface to access an internal resource. Another example could be a device that acts like a bridge to another device or bus where the initial latency to complete the access may be greater than 16 clocks. The most common types of bridges are host bus bridges, standard expansion bus bridges, and PCI-to-PCI bridges.

**Implementation Note: Using More Than One Option to Meet Initial Latency**

A combination of the different options may be used based on the access latency of a particular device. For example, a graphics controller may meet the initial latency requirements using Option 1 when accessing configuration or internal (I/O or memory mapped) registers. However, it may be required to use Option 2 or in some cases Option 3 when accessing the frame buffer.

### 3.5.1.2. Target Subsequent Latency

Target subsequent latency is the number of clocks from the assertion of **IRDY#** and **TRDY#** for one data phase to the assertion of **TRDY#** or **STOP#** for the next data phase in a burst transfer. The target is required to complete a subsequent data phase within eight clocks from the completion of the previous data phase. This requires the target to complete the data phase either by transferring data (**TRDY#** asserted), by doing target Disconnect without data (**STOP#** asserted, **TRDY#** deasserted), or by doing Target-Abort (**STOP#** asserted, **DEVSEL#** deasserted) within the target subsequent latency requirement.

In most designs, the latency to complete a subsequent data phase is known when the device is being designed. In this case, the target must manipulate **TRDY#** and **STOP#** so as to end the transaction (subsequent data phase) upon completion of data phase "N" (where N=1, 2, 3, ...), if incremental latency to data phase "N+1" is greater than eight clocks. For example, assume a PCI master read from an expansion bus takes a minimum of 15 clocks to complete each data phase. Applying the rule for N = 1, the incremental latency to data phase 2 is 15 clocks; thus, the target must terminate upon completion of

data phase 1 (i.e., a target this slow must break attempted bursts on data phase boundaries).

For designs where the latency to complete a subsequent data phase cannot be determined in advance, the target is allowed to implement a counter that causes the target to assert **STOP#** before or during the eighth clock if **TRDY#** is not asserted. If **TRDY#** is asserted before the count expires, the counter is reset and the target continues the transaction.

## 3.5.2. Master Data Latency

Master data latency is the number of clocks the master takes to assert **IRDY#** indicating it is ready to transfer data. All masters are required to assert **IRDY#** within eight clocks of the assertion of **FRAME#** on the initial data phase and within eight clocks on all subsequent data phases. Generally in the first data phase of a transaction, there is no reason for a master to delay the assertion of **IRDY#** more than one or two clocks for a write transaction. The master should never delay the assertion of **IRDY#** on a read transaction. If the master has no buffer available to store the read data, it should delay requesting use of the bus until a buffer is available. On a write transaction, the master should have the data available before requesting the bus to transfer the data. Data transfers on PCI should be done as register to register transfers to maximize performance.

## 3.5.3. Memory Write Maximum Completion Time Limit

A target may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a video screen refresh. However, a target is not permitted to terminate memory write transactions with Retry indefinitely.

After a target terminates a memory write transaction with Retry, it is required to be ready to complete at least one data phase of a memory write within a specified number of PCI clock cycles from the first Retry termination. This specified number of clock cycles is 334 clocks for systems running at 33 MHz or slower and 668 clocks for systems running at 66 MHz. This time limit, which translates to 10 microseconds at maximum frequencies (33 MHz and 66 MHz), is called the Maximum Completion Time. If a target is presented with multiple memory write requests, the Maximum Completion Time is measured from the time the first memory write transaction is terminated with Retry until the time the first data phase of any memory write to the target is completed with something other than Retry. Once a non-Retry termination has occurred, the Maximum Completion Time limit starts over again with the next Retry termination.

The actual time that the data phase completes will also depend upon when the master repeats the transaction. Targets must be designed to meet the Maximum Completion Time requirements assuming the master will repeat the memory write transaction precisely at the limit of the Maximum Completion Time.

**Implementation Note: Meeting Maximum Completion Time Limit by Restricting Use of the Device**

Some target hardware designs may not be able to process every memory write transaction within the Maximum Completion Time. An example is writing to a command queue where commands can take longer than the Maximum Completion Time to complete. Subsequent writes to such a target when it is currently processing a previous write could experience completion times that are longer than the Maximum Completion Time. Devices that take longer than the Maximum Completion Time to process some memory write transaction must restrict the usage of the device to prevent write transactions when the device cannot complete them within the Maximum Completion Time. This is typically done by the device driver and is accomplished by limiting the rate at which memory writes are issued to the device, or by reading the device to determine that a buffer is available before the write transaction is issued.

Bridge devices (Base Class = 0x06) are exempt from the Maximum Completion Time requirement for any requests that move data across the bridge. Bridge devices must follow the Maximum Completion Time requirement for transactions that address locations within (or associated with) the bridge.

The Maximum Completion Time requirement is not in effect during device initialization time, which is defined as the $2^{25}$ PCI clocks immediately following the deassertion of **RST#**.

Even though targets are required to complete memory write transactions within the Maximum Completion Time, masters cannot rely on memory write transactions completing within this time. A transaction may flow through a PCI-to-PCI bridge or be one of multiple transactions to a target. In both of these cases, the actual completion time may exceed the normal limit.

## 3.5.4. Arbitration Latency

Arbitration latency is the number of clocks from when a master asserts its **REQ#** until the bus reaches an Idle state *and* the master's **GNT#** is asserted. In a lightly loaded system, arbitration latency will generally just be the time for the bus arbiter to assert the master's **GNT#**. If a transaction is in progress when the master's **GNT#** is asserted, the master must wait the additional time for the current transaction to complete.

The total arbitration latency for a master is a function of how many other masters are granted the bus before it, and how long each one keeps the bus. The number of other masters granted the bus is determined by the bus arbiter as discussed in Section 3.4. Each master's tenure on the bus is limited by its master Latency Timer when its **GNT#** has been deasserted.

The master Latency Timer is a programmable timer in each master's Configuration Space (refer to Section 6.2.4.). It is required for each master which is capable of bursting more than two data phases. Each master's Latency Timer is cleared and suspended whenever it is not asserting **FRAME#**. When a master asserts **FRAME#**, it enables its Latency Timer to count. The master's behavior upon expiration of the Latency Timer depends on what command is being used and the state of **FRAME#** and **GNT#** when the Latency Timer expires.

- If the master deasserts **FRAME#** prior to or on the same clock that the counter expires, the Latency Timer is meaningless. The cycle terminates as it normally would when the current data phase completes.
- If **FRAME#** is asserted when the Latency Timer expires, and the command *is not* Memory Write and Invalidate, the master must initiate transaction termination when **GNT#** is deasserted, following the rules described in Section 3.3.3.1. In this case, the master has committed to the target that it will complete the current data phase and one more (the final data phase is indicated when **FRAME#** is deasserted).
- If **FRAME#** is asserted when the Latency Timer expires, the command *is* Memory Write and Invalidate, and the current data phase *is not* transferring the last DWORD of the current cacheline when **GNT#** is deasserted, the master must terminate the transaction at the end of the current cacheline (or when **STOP#** is asserted).
- If **FRAME#** is asserted when the Latency Timer expires, the command *is* Memory Write and Invalidate, and the current data phase *is* transferring the last DWORD of the current cacheline when **GNT#** is deasserted, the master must terminate the transaction at the end of the *next* cacheline. (This is required since the master committed to the target at least one more data phase, which would be the beginning of the next cacheline which it must complete, unless **STOP#** is asserted.)

In essence, the value programmed into the Latency Timer represents a minimum guaranteed number of clocks allotted to the master, after which it must surrender tenure as soon as possible after its **GNT#** is deasserted. The actual duration of a transaction (assuming its **GNT#** is deasserted) can be from a minimum of the Latency Timer value plus one clock to a maximum of the Latency Timer value plus the number of clocks required to complete an entire cacheline transfer (unless the target asserts **STOP#**).

### 3.5.4.1. Bandwidth and Latency Considerations

In PCI systems, there is a tradeoff between the desire to achieve low latency and the desire to achieve high bandwidth (throughput). High throughput is achieved by allowing devices to use long burst transfers. Low latency is achieved by reducing the maximum burst transfer length. The following discussion is provided (for a 32-bit bus) to illustrate this tradeoff.

A given PCI bus master introduces latency on PCI each time it uses the PCI bus to do a transaction. This latency is a function of the behavior of both the master and the target device during the transaction as well as the state of the master's **GNT#** signal. The bus command used, transaction burst length, master data latency for each data phase, and the Latency Timer are the primary parameters which control the master's behavior. The bus command used, target latency, and target subsequent latency are the primary parameters which control the target's behavior.

A master is required to assert its **IRDY#** within eight clocks for any given data phase (initial and subsequent). For the first data phase, a target is required to assert its **TRDY#** or **STOP#** within 16 clocks from the assertion of **FRAME#** (unless the access hits a modified cacheline in which case 32 clocks are allowed for host bus bridges). For all subsequent data phases in a burst transfer, the target must assert its **TRDY#** or **STOP#** within eight clocks. If the effects of the Latency Timer are ignored, it is a straightforward exercise to develop equations for the worst case latencies that a PCI bus master can introduce from these specification requirements.

| | | |
|---|---|---|
| **latency_max (clocks)** | **= 32 + 8 * (n-1)** | if a modified cacheline is hit (for a host bus bridge only) |
| **or** | **= 16 + 8 * (n-1)** | if not a modified cacheline |

where n is the total number of data transfers in the transaction

However, it is more useful to consider transactions that exhibit typical behavior. PCI is designed so that data transfers between a bus master and a target occur as register to register transfers. Therefore, bus masters typically do not insert wait states since they only request transactions when they are prepared to transfer data. Targets typically have an initial access latency less than the 16 (32 for modified cacheline hit for host bus bridge) clock maximum allowed. Once targets begin transferring data (complete their first data phase), they are typically able to sustain burst transfers at full rate (one clock per data phase) until the transaction is either completed by the master or the target's buffers are filled or are temporarily empty. The target can use the target Disconnect protocol to terminate the burst transaction early when its buffers fill or temporarily empty during the transaction. Using these more realistic considerations, the worst case latency equations can be modified to give a typical latency (assuming that the target's initial data phase latency is eight clocks) again ignoring the effects of the Latency Timer.

**latency_typical (clocks) = 8 + (n-1)**

If a master were allowed to burst indefinitely with a target which could absorb or source the data indefinitely, then there would be no upper bound on the latency which a master could introduce into a PCI system. However, the master Latency Timer provides a mechanism to constrain a master's tenure on the bus (when other bus masters need to use the bus).

In effect, the Latency Timer controls the tradeoff between high throughput (higher Latency Timer values) and low latency (lower Latency Timer values). Table 3-4 shows the latency for different burst length transfers using the following assumptions:

- The initial latency introduced by the master or target is eight clocks.
- There is no latency on subsequent data phases (**IRDY#** and **TRDY#** are always asserted).
- The number of data phases are powers of two because these are easy to correlate to cacheline sizes.
- The Latency Timer values were chosen to expire during the next to last data phase, which allows the master to complete the correct number of data phases.

For example, with a Latency Timer of 14 and a target initial latency of 8, the Latency Timer expires during the seventh data phase. The transaction completes with the eighth data phase.

Table 3-4: Latency for Different Burst Length Transfers

| Data Phases | Bytes Transferred | Total Clocks | Latency Timer (clocks) | Bandwidth (MB/s) | Latency (μs) |
|---|---|---|---|---|---|
| 8 | 32 | 16 | 14 | 60 | .48 |
| 16 | 64 | 24 | 22 | 80 | .72 |
| 32 | 128 | 40 | 38 | 96 | 1.20 |
| 64 | 256 | 72 | 70 | 107 | 2.16 |

**Data Phases** Number of data phases completed during transaction

**Bytes Transferred** Total number of bytes transferred during transaction (assuming 32-bit transfers)

**Total Clocks** Total number of clocks used to complete the transfer

**total_clocks = 8 + (n-1) + 1** (Idle time on bus)

**Latency Timer** Latency Timer value in clocks such that the Latency Timer expires in next to last data phase

**latency_timer = total_clocks - 2**

**Bandwidth** Calculated bandwidth in MB/s

**bandwidth = bytes_transferred / (total clocks * 30 ns)**

**Latency** Latency in microseconds introduced by transaction

**latency = total clocks * 30 ns**

Table 3-4 clearly shows that as the burst length increases, the amount of data transferred increases. Note: The amount of data doubles between each row in the table, while the latency increases by less than double. The amount of data transferred between the first row and the last row increases by a factor of 8, while the latency increases by a factor of 4.5. The longer the transaction (more data phases), the more efficiently the bus is being used. However, this increase in efficiency comes at the expense of larger buffers.

## 3.5.4.2. Determining Arbitration Latency

Arbitration latency is the number of clocks a master must wait after asserting its **REQ#** before it can begin a transaction. This number is a function of the arbitration algorithm of the system; i.e., the sequence in which masters are given access to the bus and the value of the Latency Timer of each master. Since these factors will vary from system to system, the best an individual master can do is to pick a configuration that is considered the typical case and apply the latency discussion to it to determine the latency a device will experience.

Arbitration latency is also affected by the loading of the system and how efficient the bus is being used. The following two examples illustrate a lightly and heavily loaded system where the bus (PCI) is 32-bit. The lightly loaded example is the more typical case of systems today, while the second is more of a theoretical maximum.

**Lightly Loaded System**

For this example, assume that no other **REQ#**s are asserted and the bus is either in the Idle state or that a master is currently using the bus. Since no other **REQ#**s are asserted, as soon as Agent A's **REQ#** is asserted, the arbiter will assert its **GNT#** on the next evaluation of the **REQ#** lines. In this case, Agent A's **GNT#** will be asserted within a few clocks. Agent A gains access to the bus when the bus is in the Idle state (assuming its **GNT#** is still active).

**Heavily Loaded System**

This example will use the arbiter described in the implementation note in Section 3.4. Assume that all agents have their **REQ#** lines asserted and all want to transfer more data than their Latency Timers allow. To start the sequence, assume that the next bus master is Agent A on level 1 and Agent X on level 2. In this example, Agent A has a very small number of clocks before it gains access to the bus, while Agent Z has the largest number. In this example, Agents A and B each get a turn before an Agent at Level 2. Therefore, Agents A and B each get three turns on the bus, and Agents X and Y each get one turn before Agent Z gets a turn. Arbitration latency (in this example) can be as short as a few clocks for Agent A or (assuming a Latency Timer of 22 clocks) as long as 176 clocks (8 masters * 22 clocks/master) for Agent Z. Just to keep this in perspective, the heavily loaded system is constantly moving about 90 MB/s of data (assuming target initial latency of eight clocks and target subsequent latency of one clock).

As seen in the example, a master experiences its maximum arbitration latency when all the other masters use the bus up to the limits of their Latency Timers. The probability of this happening increases as the loading of the bus increases. In a lightly loaded system, fewer masters will need to use the bus or will use it less than their Latency Timer would allow, thus allowing quicker access by the other masters.

How efficiently each agent uses the bus will also affect average arbitration latencies. The more wait states a master or target inserts on each transaction, the longer each transaction will take, thus increasing the probability that each master will use the bus up to the limit of its Latency Timer.

The following examples illustrate the impact on arbitration latency as the efficiency of the bus goes down due to wait states being inserted. In both examples, the system has a single arbitration level, the Latency Timer is set to 22 and there are five masters that have data to move. A Latency Timer of 22 allows each master to move a 64-byte cacheline if initial latency is only eight clocks and subsequent latency is one clock. The high bus efficiency example illustrates that the impact on arbitration latency is small when the bus is being used efficiently.

**System with High Bus Efficiency**

In this example, each master is able to move an entire 64-byte cacheline before its respective Latency Timer expires. This example assumes that each master is ready to transfer another cacheline just after it completes its current transaction. In this example, the Latency Timer has no affect. It takes the master

[(1 idle clock) + (8 initial **TRDY#** clocks)+ (15 subsequent **TRDY#** clocks)]
* 30 ns/clock = 720 ns

to complete each cacheline transfer.

If all five masters use the same number of clocks, then each master will have to wait for the other four, or

720 ns/master * 4 other masters = 2.9 µs

between accesses. Each master moves data at about 90 MB/s.

The Low Bus Efficiency example illustrates the impact on arbitration latency as a result of the bus being used inefficiently. The first effect is that the Latency Timer expires. The second effect is that is takes two transactions to complete a single cacheline transfer which causes the loading to increase.

**System with Low Bus Efficiency**

This example keeps the target initial latency the same but increases the subsequent latency (master or target induced) from 1 to 2. In this example, the Latency Timer will expire before the master has transferred the full 64-byte cacheline. When the Latency Timer expires, **GNT#** is deasserted, and **FRAME#** is asserted, the master must stop the transaction prematurely and completes the final two data phases it has committed to complete (unless a MWI command in which case it completes the current cacheline). Each master's tenure on the bus would be

[(1 idle clock) + (22 Latency Timer clocks)+
(2 * 2 subsequent **TRDY#** clocks)]
* 30 ns/clock = 810 ns

and each master has to wait

810 ns/master * 4 other masters = 3.2 µs

between accesses. However, the master only took slightly more time than the High Bus Efficiency example, but only completed nine data phases (36 bytes, just over half a cacheline) instead of 16 data phases. Each master moves data at only about 44 MB/s.

The arbitration latency in the Low Bus Efficiency example is 3 µs instead of 2.9 µs as in the High Bus Efficiency example; but it took the master two transactions to complete the transfer of a single cacheline. This doubled the loading of the system without increasing

the data throughput. This resulted from simply adding a single wait state to each data phase.

Also, note that the above description assumes that all five masters are in the same arbitration level. When a master is in a lower arbitration level or resides behind a PCI-to-PCI bridge, it will experience longer latencies between accesses when the primary PCI bus is in use.

The maximum limits of a target and master data latency in this specification are provided for instantaneous conditions while the recommendations are used for normal behavior. An example of an instantaneous condition is when the device is unable to continue completing a data phase on each clock. Rather than stopping the transfer (introducing the overhead of re-arbitration and target initial latency), the target would insert a couple of wait states and continue the burst by completing a data phase on each clock. The maximum limits are not intended to be used on every data phase, but rather on those rare occasions when data is temporarily unable to transfer.

The following discussion assumes that devices are compliant with the specification and have been designed to minimize their impact on the bus. For example, a master is required to assert **IRDY#** within eight clocks for all data phases; however, it is recommended that it assert **IRDY#** within one or two clocks.

**Example of a System**

The following system configuration and the bandwidth each device requires are generous and exceed the needs of current implementations. The system that will be used for a discussion about latency is a workstation comprised of:

Host bus bridge (with integrated memory controller)
Graphics device (VGA and enhanced graphics)
Frame grabber (for video conferencing)
LAN connection
Disk (a single spindle, IDE or SCSI)
Standard expansion bus bridge (PCI to ISA)
A PCI-to-PCI bridge for providing more than three add-in slots

The graphics controller is capable of sinking 50 MB/s. This assumes that the host bus bridge generates 30 MB/s and the frame grabber generates 20 MB/s.

The LAN controller requires only about 4 MB/s (100 Mb) on average (workstation requirements) and is typically much less.

The disk controller can move about 5 MB/s.

The standard expansion bus provides a cost effective way of connecting standard I/O controllers (i.e., keyboard, mouse, serial, parallel ports, etc.) and masters on this bus place a maximum of about 4 MB/s (aggregate plus overhead) on PCI and will decrease in future systems.

The PCI-to-PCI bridge, in and of itself, does not use PCI bandwidth, but a place holder of 9 MB/s is allocated for devices that reside behind it.

The total bandwidth needs of the system is about 72 MB/s (50 + 4 + 5 + 4 + 9) if all devices want to use the bus at the same time.

To show that the bus can handle all the devices, these bandwidth numbers will be used in the following discussion. The probability of all devices requiring use of the bus at the same time is extremely low, and the typical latency will be much lower than the worst cases number discussed. For this discussion, the typical numbers used are at a steady state condition where the system has been operating for a while and not all devices require access to the bus at the same time.

Table 3-5 lists the requirements of each device in the target system and how many transactions each device must complete to sustain its bandwidth requirements within 10 µs time slices.

The first column identifies the device generating the data transfer.

The second column is the total bandwidth the device needs.

The third column is the approximate number of bytes that need to be transferred during this 10 µs time slice.

The fourth column is the amount of time required to move the data.

The last column indicates how many different transactions that are required to move the data. This assumes that the entire transfer cannot be completed as a single transaction.

**Table 3-5: Example System**

| Device | Bandwidth (MB/s) | Bytes/10 µs | Time Used (µs) | Number of Transactions per Slice | Notes |
|---|---|---|---|---|---|
| Graphics | 50 | 500 | 6.15 | 10 | 1 |
| LAN | 4 | 40 | 0.54 | 1 | 2 |
| Disk | 5 | 50 | 0.63 | 1 | 3 |
| ISA bridge | 4 | 40 | 0.78 | 2 | 4 |
| PCI-to PCI bridge | 9 | 90 | 1.17 | 2 | 5 |
| Total | 72 | 720 | 9.27 | 16 | |

Notes:

1. Graphics is a combination of host bus bridge and frame grabber writing data to the frame buffer. The host moves 300 bytes using five transactions with 15 data phases each, assuming eight clocks of target initial latency. The frame grabber moves 200 bytes using five transactions with 10 data phases each, assuming eight clocks of target initial latency.
2. The LAN uses a single transaction with 10 data phases with eight clocks of target initial latency.
3. The disk uses a single transaction with 13 data phases with eight clocks of target initial latency.
4. The ISA bridge uses two transactions with five data phases each with eight clocks of target initial latency.
5. The PCI-to-PCI bridge uses two transactions. One transaction is similar to the LAN and the second is similar to the disk requirements.

If the targeted system only needs full motion video or a frame grabber but not both, then replace the Graphics row in Table 3-5 with the appropriate row in Table 3-6. In either case, the total bandwidth required on PCI is reduced.

**Table 3-6: Frame Grabber or Full Motion Video Example**

| Device | Bandwidth (MB/s) | Bytes/10 µs | Time Used (µs) | Number of Transactions per Slice | Notes |
|---|---|---|---|---|---|
| Host writing to the frame buffer | 40 | 400 | 4.2 | 5 | 1 |
| Frame grabber | 20 | 200 | 3.7 | 5 | 2 |

Notes

1. The host uses five transactions with 20 data phases each, assuming eight clocks of target initial latency.
2. The frame grabber uses five transactions with 10 data phases each, assuming eight clocks of target initial latency.

The totals for Table 3-5 indicate that within a 10 µs window all the devices listed in the table move the data they required for that time slice. In a real system, not all devices need to move data all the time. But they may be able to move more data in a single transaction. When devices move data more efficiently, the latency each device experiences is reduced.

If the above system supported the arbiter illustrated in the System Arbitration Algorithm Implementation Note (refer to Section 3.4.), the frame grabber (or graphics device when it is a master) and the PCI-to-PCI bridge would be put in the highest level. All other devices would be put in the lower level (i.e., level two). Table 3-5 shows that if all devices provide 10 µs of buffering, they would not experience underruns or overruns. However, for devices that move large blocks of data and are generally given higher priority in a system, then a latency of 3 µs is reasonable. (When only two agents are at the highest level, each experiences about 2 µs of delay between transactions. The table assumes that the target is able to consume all data as a single transaction.)

## 3.5.4.3. Determining Buffer Requirements

Each device that interfaces to the bus needs buffering to match the rate the device produces or consumes data with the rate that it can move data across the bus. The size of buffering can be determined by several factors based on the functionality of the device and the rate at which it handles data. As discussed in the previous section, the arbitration latency a master experiences and how efficiently data is transferred on the bus will affect the amount of buffering a device requires.

In some cases, a small amount of buffering is required to handle errors, while more buffering may give better bus utilization. For devices which do not use the bus very much (devices which rarely require more than 5 MB/s), it is recommended that a minimum of four DWORDs of buffering be supported to ensure that transactions on the bus are done with reasonable efficiency. Moving data as entire cachelines is the preferred transfer size. Transactions less than four DWORDs in length are inefficient

and waste bus bandwidth. For devices which use the bus a lot (devices which frequently require more than 5 MB/s), it is recommended that a minimum of 32 DWORDs of buffering be supported to ensure that transactions on the bus are done efficiently. Devices that do not use the bus efficiently will have a negative impact on system performance and a larger impact on future systems.

While these recommendations are minimums, the real amount of buffering a device needs is directly proportional to the difficulty required to recover from an underrun or overrun. For example, a disk controller would provide sufficient buffering to move data efficiently across PCI, but would provide no additional buffering for underruns and overruns (since they will not occur). When data is not available to write to the disk, the controller would just wait until data is available. For reads, when a buffer is not available, it simply does not accept any new data.

A frame grabber must empty its buffers before new data arrives or data is destroyed. For systems that require good video performance, the system designer needs to provide a way for that agent to be given sufficient bus bandwidth to prevent data corruption. This can be accomplished by providing an arbiter that has different levels and/or adjusting the Latency Timer of other masters to limit their tenure on the bus.

The key for future systems is to have all devices use the bus as efficiently as possible. This means to move as much data as possible (preferably several cachelines) in the smallest number of clocks (preferably one clock subsequent latency). As devices do this, the entire system experiences greater throughput and lower latencies. Lower latencies allow smaller buffers to be provided in individual devices. Future benchmarks will allow system designers to distinguish between devices that use the bus efficiently and those that do not. Those that do will enable systems to be built that meet the demands of multimedia systems.

## 3.6. Other Bus Operations

### 3.6.1. Device Selection

**DEVSEL#** is driven by the target of the current transaction as shown in Figure 3-17 to indicate that it is responding to the transaction. **DEVSEL#** may be driven one, two, or three clocks following the address phase. Each target indicates the **DEVSEL#** timing it uses in its Configuration Space Status register described in Section 6.2.3. **DEVSEL#** must be asserted with or prior to the edge at which the target enables its **TRDY#**, **STOP#**, and data if a read transaction. In other words, a target must assert **DEVSEL#** (claim the transaction) before or coincident with signaling any other target response. Once **DEVSEL#** has been asserted, it cannot be deasserted until the last data phase has completed, except to signal Target-Abort. Refer to Section 3.3.3.2. for more information.

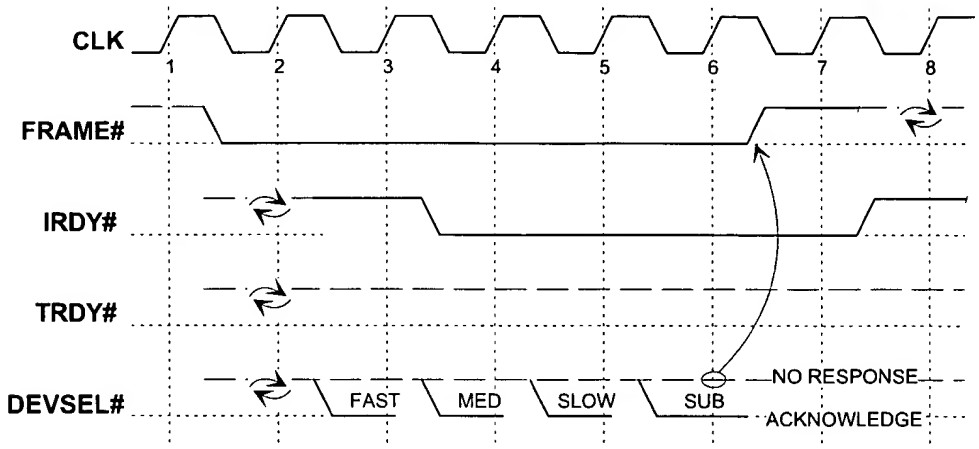



**Figure 3-17: DEVSEL# Assertion**

If no agent asserts **DEVSEL#** within three clocks of **FRAME#**, the agent doing subtractive decode may claim and assert **DEVSEL#**. If the system does not have a subtractive decode agent, the master never sees **DEVSEL#** asserted and terminates the transaction per the Master-Abort mechanism (refer to Section 3.3.3.1.).

A target must do a full decode before driving/asserting **DEVSEL#**, or any other target response signal. It is illegal to drive **DEVSEL#** prior to a complete decode and then let the decode combinationally resolve on the bus. (This could cause contention.) A target must qualify the **AD** lines with **FRAME#** before **DEVSEL#** can be asserted on commands other than configuration. A target must qualify **IDSEL** with **FRAME#** and **AD[1::0]** before **DEVSEL#** can be asserted on a configuration command.

It is expected that most (perhaps all) target devices will be able to complete a decode and assert **DEVSEL#** within one or two clocks of **FRAME#** being asserted (fast and medium in the figure).

Accordingly, the subtractive decode agent may provide an optional device dependent configuration register that can be programmed to pull in by one or two clocks the edge at which it asserts **DEVSEL#**, allowing faster access to the expansion bus. Use of such an option is limited by the slowest positive decode agent on the bus.

If the first byte addressed by the transaction maps into the target's address range, it asserts **DEVSEL#** to claim the access. But if the master attempts to continue the burst transaction across the resource boundary, the target is required to signal Disconnect.

When a target claims an I/O access and the byte enables indicate one or more bytes of the access are outside the target's address range, it must signal Target-Abort. (Refer to Section 3.3.3.2. for more information.) To deal with this type of I/O access problem, a subtractive decode device (expansion bus bridge) may do one of the following:

- Do positive decode (by including a byte map) on addresses for which different devices share common DWORDs, additionally using byte enables to detect this problem and signal Target-Abort.
- Pass the full access to the expansion bus, where the portion of the access that cannot be serviced will quietly drop on the floor. (This occurs only when the first addressed target resides on the expansion bus and the other is on PCI.)

## 3.6.2. Special Cycle

The Special Cycle command provides a simple message broadcast mechanism on PCI. In addition to communicating processor status (as is done on Intel processor buses), it may also be used for logical sideband signaling between PCI agents, when such signaling does not require the precise timing or synchronization of physical signals.

A good paradigm for the Special Cycle command is that of a "logical wire" which only signals single clock pulses; i.e., it can be used to set and reset flip flops in real time implying that delivery is guaranteed. This allows the designer to define necessary sideband communication without requiring additional pins. As with sideband signaling in general, implementation of Special Cycle command support is optional.

The Special Cycle command contains no explicit destination address, but is broadcast to all agents on the same bus segment. Each receiving agent must determine whether the message is applicable to it. PCI agents will never assert **DEVSEL#** in response to a Special Cycle command.

Note: Special Cycle commands do not cross PCI-to-PCI bridges. If a master desires to generate a Special Cycle command on a specific bus in the hierarchy, it must use a Type 1 configuration write command to do so. Type 1 configuration write commands can traverse PCI-to-PCI bridges in both directions for the purpose of generating Special Cycle commands on any bus in the hierarchy and are restricted to a single data phase in length. However, the master must know the specific bus on which it desires to generate the Special Cycle command and cannot simply do a broadcast to one bus and expect it to propagate to all buses. Refer to Section 3.2.2.3.1. for more information.

A Special Cycle command may contain optional, message dependent data, which is not interpreted by the PCI sequencer itself, but is passed, as necessary, to the hardware application connected to the PCI sequencer. In most cases, explicitly addressed messages should be handled in one of the three physical address spaces on PCI and not with the Special Cycle command.

Using a message dependent data field can break the logical wire paradigm mentioned above and create delivery guarantee problems. However, since targets only accept messages they recognize and understand, the burden is placed on them to fully process the message in the minimum delivery time (six bus clocks) or to provide any necessary buffering for messages they accept. Normally this buffering is limited to a single flip-flop. This allows delivery to be guaranteed. In some cases, it may not be possible to buffer or process all messages that could be received. In this case, there is no guarantee of delivery.

A Special Cycle command is like any other bus command where there is an address phase and a data phase. The address phase starts like all other commands with the assertion of **FRAME#** and completes like all other commands when **FRAME#** and **IRDY#** are deasserted. The uniqueness of this command compared to the others is that no agent responds with the assertion of **DEVSEL#** and the transaction concludes with a Master-Abort termination. Master-Abort is the normal termination for Special Cycle transactions and no errors are reported for this case of Master-Abort termination. This command is basically a broadcast to all agents, and interested agents accept the command and process the request.

The address phase contains no valid information other than the command field. There is no explicit address; however, **AD[31::00]** are driven to a stable level and parity is generated. During the data phase, **AD[31::00]** contain the message type and an optional

data field. The message is encoded on the least significant 16 lines, namely **AD[15::00]**. The optional data field is encoded on the most significant 16 lines, namely **AD[31::16]**, and is not required on all messages. The master of a Special Cycle command can insert wait states like any other command while the target cannot (since no target claimed the access by asserting **DEVSEL#**). The message and associated data are only valid on the first clock **IRDY#** is asserted. The information contained in, and the timing of, subsequent data phases are message dependent. When the master inserts a wait state or performs multiple data phases, it must extend the transaction to give potential targets sufficient time to process the message. This means the master must guarantee the access will not complete for at least four clocks (may be longer) after the last valid data completes. For example, a master keeps **IRDY#** deasserted for two clocks for a single data phase Special Cycle command. Because the master inserted wait states, the transaction cannot be terminated with Master-Abort on the fifth clock after **FRAME#** (the clock after subtractive decode time) like usual, but must be extended at least an additional two clocks. When the transaction has multiple data phases, the master cannot terminate the Special Cycle command until at least four clocks after the last valid data phase. Note: The message type or optional data field will indicate to potential targets the amount of data to be transferred. The target must latch data on the first clock **IRDY#** is asserted for each piece of data transferred.

During the address phase, **C/BE[3::0]#** = 0001 (Special Cycle command) and **AD[31::00]** are driven to random values and must be ignored. During the data phase, **C/BE[3::0]#** are asserted and **AD[31::00]** are as follows:

**AD[15::00]** Encoded message

**AD[31::16]** Message dependent (optional) data field

The PCI bus sequencer starts this command like all others and terminates it with a Master-Abort. The hardware application provides all the information like any other command and starts the bus sequencer. When the sequencer reports that the access terminated with a Master-Abort, the hardware application knows the access completed. In this case, the Received Master Abort bit in the configuration Status register (Section 6.2.3.) must not be set. The quickest a Special Cycle command can complete is five clocks. One additional clock is required for the turnaround cycle before the next access. Therefore, a total of six clocks is required from the beginning of a Special Cycle command to the beginning of another access.

There are a total of 64 K messages. The message encodings are defined and described in Appendix A.

## 3.6.3. Address/Data Stepping

The ability of an agent to spread assertion of qualified signals over several clocks is referred to as *stepping*. This notion allows an agent with "weak" output buffers to drive a set of signals to a valid state over several clocks (*continuous stepping*), thereby reducing the ground current load generated by each buffer. An alternative approach allows an agent with "strong" output buffers to drive a subset of them on each of several clock edges until they are all driven (*discrete stepping*), thereby reducing the number of signals that must be switched simultaneously. All agents must be able to handle address and data stepping while generating it is optional. Refer to Section 4.2.4. for conditions associated with indeterminate signal levels on the rising edge of **CLK**.

Either continuous or discrete stepping allows an agent to trade off performance for cost (fewer power/ground pins). When using the continuous stepping approach, care must be taken to avoid mutual coupling between critical control signals that must be sampled on each clock edge and the stepped signals that may be transitioning on a clock edge. Performance critical peripherals should apply this "permission" sparingly.

Stepping is only permitted on **AD[31::00]**, **AD[63::32]**, **PAR**, **PAR64#** (for 64-bit data transfers but not for the DAC command), and **IDSEL** pins, because they are always qualified by control signals; i.e., these signals are only considered valid on clock edges for which they are qualified. **AD**s are qualified by **FRAME#** in address phases and by **IRDY#** or **TRDY#** in data phases (depending on which direction data is being transferred). **PAR** is implicitly qualified on each clock after which **AD** was qualified. **IDSEL** is qualified by the combination of **FRAME#** and a decoded Type 0 configuration command.

Figure 3-18 illustrates a master delaying the assertion of **FRAME#** until it has successfully driven all **AD** lines. The master is both permitted and required to drive **AD** and **C/BE#** once ownership has been granted and the bus is in the Idle state. But it may take multiple clocks to drive a valid address before asserting **FRAME#**. However, by delaying assertion of **FRAME#**, the master runs the risk of losing its turn on the bus. As with any master, **GNT#** must be asserted on the rising clock edge before **FRAME#** is asserted. If **GNT#** were deasserted, on the clock edges marked "A", the master is required to immediately tri-state its signals because the arbiter has granted the bus to another agent. (The new master would be at a higher priority level.) If **GNT#** were deasserted on the clock edges marked "B" or "C", **FRAME#** will have already been asserted and the transaction continues.

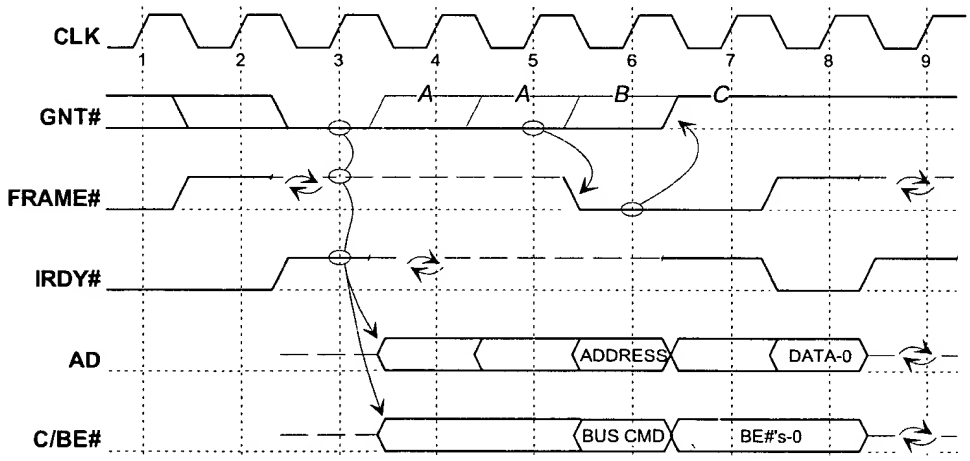



**Figure 3-18: Address Stepping**

## 3.6.4. Interrupt Acknowledge

The PCI bus supports an Interrupt Acknowledge cycle as shown in Figure 3-19. This figure illustrates an x86 Interrupt Acknowledge cycle on PCI where a single byte enable is asserted and is presented only as an example. In general, the byte enables determine which bytes are involved in the transaction. During the address phase, **AD[31::00]** do not contain a valid address but must be driven with stable data, **PAR** is valid, and parity may be checked. An Interrupt Acknowledge transaction has no addressing mechanism and is implicitly targeted to the interrupt controller in the system. As defined in the *PCI-to-PCI Bridge Architecture Specification*, the Interrupt Acknowledge command is not forwarded to another PCI segment. The Interrupt Acknowledge cycle is like any other transaction in that **DEVSEL#** must be asserted one, two, or three clocks after the assertion of **FRAME#** for positive decode and may also be subtractively decoded by a standard expansion bus bridge. Wait states can be inserted and the request can be terminated, as discussed in Section 3.3.3.2. The vector must be returned when **TRDY#** is asserted.

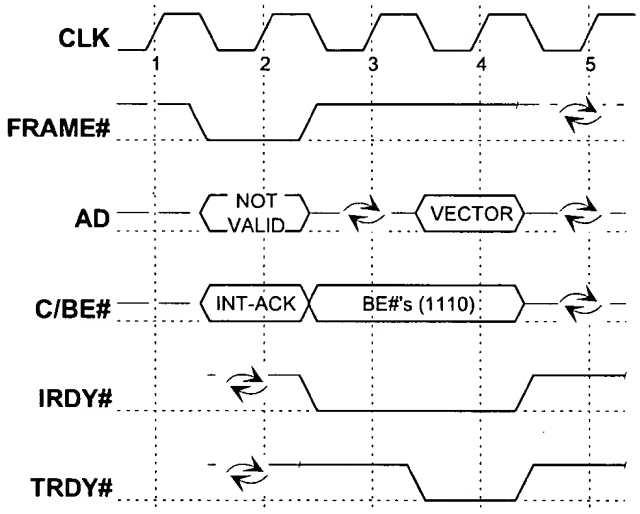



**Figure 3-19: Interrupt Acknowledge Cycle**

Unlike the traditional 8259 dual cycle acknowledge, PCI runs a single cycle acknowledge. Conversion from the processor's two cycle format to the PCI one cycle format is easily done in the bridge by discarding the first Interrupt Acknowledge request from the processor.

## 3.7. Error Functions

PCI provides for parity and other system errors to be detected and reported. A single system may include devices that have no interest in errors (particularly parity errors) and agents that detect, signal, and recover from errors. PCI error reporting allows agents that recover from parity errors to avoid affecting the operation of agents that do not. To allow this range of flexibility, the generation of parity is required on all transactions by all agents. The detection and reporting of errors is generally required, with limited exclusions for certain classes of PCI agents as listed in Section 3.7.2.

## 3.7.1. Parity Generation

Parity on PCI provides a mechanism to determine for each transaction if the master is successful in addressing the desired target and if data transfers correctly between them. To ensure that the correct bus operation is performed, the four command lines are included in the parity calculation. To ensure that correct data is transferred, the four byte enables are also included in the parity calculation. The agent that is responsible for driving **AD[31::00]** on any given bus phase is also responsible for driving even parity on **PAR**. The following requirements also apply when the 64-bit extensions are used (refer to Section 3.8. for more information).

During address and data phases, parity covers **AD[31::00]** and **C/BE[3::0]#** lines regardless of whether or not all lines carry meaningful information. Byte lanes not actually transferring data are still required to be driven with stable (albeit meaningless) data and are included in the parity calculation. During configuration, Special Cycle or Interrupt Acknowledge transactions some (or all) address lines are not defined but are required to be driven to stable values and are included in the parity calculation.

Parity is generated according to the following rules:

- Parity is calculated the same on all PCI transactions regardless of the type or form.
- The number of "1"s on **AD[31::00]**, **C/BE[3::0]#**, and **PAR** equals an even number.
- Parity generation is not optional; it must be done by all PCI-compliant devices.

On any given bus phase, **PAR** is driven by the agent that drives **AD[31::00]** and lags the corresponding address or data by one clock. Figure 3-20 illustrates both read and write transactions with parity. The master drives **PAR** for the address phases on clocks 3 and 7. The target drives **PAR** for the data phase on the read transaction (clock 5) and the master drives **PAR** for the data phase on the write transaction (clock 8). Note: Other than the one clock lag, **PAR** behaves exactly like **AD[31::00]** including wait states and turnaround cycles.

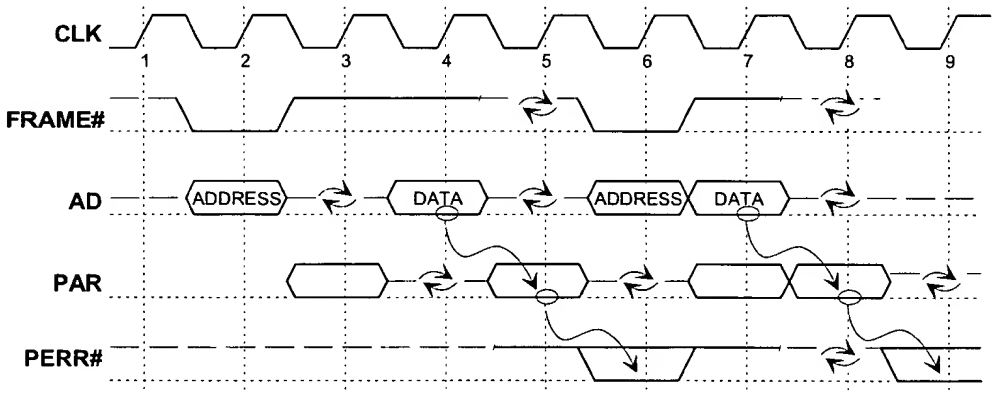



**Figure 3-20: Parity Operation**

## 3.7.2. Parity Checking

Parity must be checked to determine if the master successfully addressed the desired target and if data transferred correctly. All devices are required to check parity, except devices in the following two classes for which parity checking is optional:

- Devices that are designed exclusively for use on the motherboard or planar; e.g., chip sets. System vendors have control over the use of these devices since they will never appear on expansion boards.
- Devices that never deal with or contain or access any data that represents permanent or residual system or application state; e.g., human interface and video/audio devices. These devices only touch data that is a temporary representation (e.g., pixels) of permanent or residual system or application state. Therefore, they are not prone to create system integrity problems in the event of undetected failure.

## 3.7.3. Address Parity Errors

A device is said to have detected an address parity error if the device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle.

- If a device detects an address parity error, in some cases, it will assert **SERR#** (refer to Section 3.7.4.2.), and in all cases it will set the Detected Parity Error bit (Status register, bit 15) (refer to Section 3.7.4.4.).

If a device detects an address parity error, and the device's Parity Error Response bit (Command register, bit 6) is set, and the device's address decoder indicates that the device is selected, the device must do one of the following:

- claim the transaction and terminate it as if there was no address/command error.
- claim the transaction and terminate with Target-Abort.
- not claim the transaction and let it terminate with Master-Abort.

An error in the address phase of a transaction may affect any or all of the address bits, the command bits, and the parity bit. Since devices monitoring the transaction cannot determine which bits are actually in error, use of a transaction that contained an address parity error may cause unpredictable results.

The target is not allowed to claim a transaction and terminate it with Retry solely because of an address parity error or a write[21] data parity error. However, the occurrence of a parity error does not prevent the target from terminating the transaction with Retry for other reasons.

## 3.7.4. Error Reporting

PCI provides for the detection and signaling of two kinds of errors: data parity errors and other system errors. It is intended that data parity errors be reported up through the access and device driver chain whenever possible. This error reporting chain from target

[21] Targets check data parity only on write transactions.

to bus master to device driver to device manager to operating system is intended to allow error recovery options to be implemented at any level. Since it is generally not possible to associate system errors with a specific access chain, they are reported via a separate system error signal (refer to Section 3.7.4.2.).

PCI devices are enabled to report data parity errors by the Parity Error Response bit (bit 6 of the Command register). This bit is required in all devices except those not required to check parity (refer to Section 3.7.2.). If the Parity Error Response bit is set, devices must respond to and report data parity errors for all bus operations (except those that occur during a Special Cycle transaction). If the Parity Error Response bit is cleared, an agent that detects a data parity error must ignore the error and complete the transaction as though parity was correct. In this case, no special handling of the data parity error can occur.

Two signals (pins) and two status bits are used in the PCI error reporting scheme. Each will be discussed separately.

### 3.7.4.1. Data Parity Error Signaling on PERR#

**PERR#** is used for signaling data parity errors on all transactions except Special Cycle transactions. Data parity errors that occur during a Special Cycle transaction are reported on **SERR#** as described in Section 3.7.4.2. **PERR#** is required for all devices except those not required to check parity (refer to Section 3.7.2.).

If parity error response is enabled (bit 6 of the Command register is set) and a data parity error is detected by a master during a read transaction, the master must assert **PERR#**. If parity error response is enabled (bit 6 of the Command register is set) and a data parity error is detected by a target during a write transaction, the target must assert **PERR#**. Masters use this information to record the occurrence of the error for the device driver. **PERR#** is both an input and output signal for a master and only an output signal for a target.

A device asserting **PERR#** must do so two clocks after the completion of a data phase in which an error occurs, as shown in Figure 3-20. If the receiving agent inserts wait states, that agent is permitted to assert **PERR#** as soon as a data parity error is detected. In other words, if the target is inserting wait states during a write transaction, the target is permitted to assert **PERR#** two clocks after data is valid (**IRDY#** asserted) but before the data transfers (**TRDY#** is also asserted). If the master is inserting wait states during a read transaction, the master is permitted to assert **PERR#** two clocks after data is valid (**TRDY#** is asserted) but before the data transfers (**IRDY#** is also asserted). Once **PERR#** is asserted, it must remain asserted until two clocks following the completion of the data phase (**IRDY#** and **TRDY#** both asserted). Note that the master is required to provide valid byte enables during every clock cycle of every data phase for both read and write transactions independent of **IRDY#**.

If a master asserts **PERR#** prior to completion of a read data phase, it must eventually assert **IRDY#** to complete the data phase. If a target asserts **PERR#** prior to completion of a write data phase, it must eventually assert **TRDY#** to complete the data phase. The target cannot terminate the data phase by signaling Retry, Disconnect without data, or Target-Abort after signaling **PERR#**. A master knows a data parity error occurred on a write data phase anytime **PERR#** is asserted, which may be prior to the completion of the data phase. But the master only knows the data phase was error free two clocks following the completion of the data phase.

Both masters and targets are permitted either to continue a burst transaction or stop it after detecting a data parity error. During a burst transaction in which multiple data phases are completed without intervening wait states, **PERR#** will be qualified on multiple consecutive clocks accordingly and may be asserted in any or all of them.

**PERR#** is a sustained tri-state signal that is bused to all PCI agents. It must be actively driven to the correct value on each qualified clock edge by the agent receiving the data. At the end of each bus operation, **PERR#** must actively be driven high for one clock period by the agent receiving data, starting two clocks after the **AD** bus turnaround cycle (e.g., clock 7 in Figure 3-20). The **PERR#** turnaround cycle occurs one clock later (clock 8 in Figure 3-20). **PERR#** cannot be driven (enabled) for the current transaction until at least three clocks after the address phase (which is one clock long for single address cycles and two clocks long for dual address cycles). Note that the target of a write transaction must not drive any signal until after asserting **DEVSEL#**; for example, for decode speed "slow" the target must not drive **PERR#** until four clocks after the address phase.

### 3.7.4.2. Other Error Signaling on SERR#

If a device is enabled to assert **SERR#** (i.e., SERR# Enable, bit 8 of the Command register, is set), and the device's Parity Error Response bit (Command register, bit 6) is set, the device must assert **SERR#** if any of the following conditions occurs:

- The device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle (regardless of the intended target).
- The device monitors Special Cycle transactions, and the Special Cycles bit (Command register, bit 3) is set, and the device's parity checking logic detects a data parity error.
- The device is the master of a Message Signaled Interrupt and the transaction competes with Master-Abort or Target-Abort or the target asserts **PERR#** (refer to Section 6.8.2.1.).

**SERR#** may optionally be used to report other internal errors that might jeopardize system or data integrity. It must be assumed, however, that signaling on **SERR#** will generate a critical system interrupt (e.g., NMI or Machine Check) and is, therefore, fatal. Consequently, care should be taken in using **SERR#** to report non-parity or system errors.

**SERR#** is required for all devices except those not required to check parity (refer to Section 3.7.2.). **SERR#** is an open drain signal that is wire-ORed with all other PCI agents and, therefore, may be simultaneously driven by multiple agents. An agent reporting an error on **SERR#** drives it active for a single clock and then tri-states it. (Refer to Section 2.2.5. for more details.) Since open drain signaling cannot guarantee stable signals on every rising clock edge, once **SERR#** is asserted its logical value must be assumed to be indeterminate until the signal is sampled in the deasserted state on at least two successive rising clock edges.

### 3.7.4.3. Master Data Parity Error Status Bit

The Master Data Parity Error bit (Status register, bit 8) must be set by the master if its Parity Error Response bit (Command register, bit 6) is set and either of the following two conditions occurs:

- The master detects a data parity error on a read transaction.
- The master samples **PERR#** asserted on a write transaction.

If the Parity Error Response bit is cleared, the master must not set the Master Data Parity Error bit, even if the master detects a parity error or the target asserts **PERR#**.

Targets never set the Master Data Parity Error bit.

### 3.7.4.4. Detected Parity Error Status Bit

The Detected Parity Error bit (Status register, bit 15) must be set by a device whenever its parity checking logic detects a parity error, regardless of the state the Parity Error Response bit (bit 6 of the command register). The Detected Parity Error bit is required to be set by the device when any of the following conditions occurs:

- The device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle.
- The device's parity checking logic detects a data parity error and the device is the target of a write transaction.
- The device's parity checking logic detects a data parity error and the device is the master of a read transaction.

## 3.7.5. Delayed Transactions and Data Parity Errors

This section presents additional requirements for error handling that are unique to a target completing a transaction as a Delayed Transaction. Data parity error requirements presented in previous sections apply to Delayed Transactions as well.

A data parity error can occur during any of the three steps of a Delayed Transaction, the master request step, the target completion step, or the master completion step (refer to Section 3.3.3.3.1.). The requirements for handling the error vary depending upon the step in which the error occurred. Errors that occur during the target completion phase are specific to the target device and are handled in a device-specific manner (not specified here).[22] Device behavior for errors that occur during the master request step or master completion step depend upon whether the Delayed Transaction is a read[23] or a write[24] transaction.

---

[22] If the actual target resides on a PCI bus segment generated by a PCI-to-PCI bridge, the target completion phase occurs across a PCI bus segment. In this case, the *PCI-to-PCI Bridge Architecture Specification* details additional requirements for error handling during the target completion phase of a read Delayed Transaction.

[23] Memory Read, Memory Read Line, Memory Read Multiple, Configuration Read, I/O Read, or Interrupt Acknowledge.

[24] Configuration Write or I/O Write, but *never* Memory Write and Invalidate or Memory Write.

During a read transaction, the target device sources the data, and parity is not valid until **TRDY#** is asserted. Therefore, a data parity error cannot occur during the master request phase or any subsequent reattempt by the master that is terminated with Retry. During the master completion step of read transaction, the target sources data and data parity and the master checks parity and conditionally asserts **PERR#** as for any other (not delayed) transaction (refer to Section 3.7.4.).

During a write transaction, the master sources the write data and must assert **IRDY#** when the data is valid independent of the response by the target device (refer to Section 3.2.1.). Therefore, a data parity error may occur both in the master request and the master completion steps. In addition, it is possible for a data parity error to be either constant (i.e., the same error occurs each time the master repeats the transaction) or transient (i.e., the error occurs on some but not other repetitions of the transaction by the master). The data parity error reporting methods for write Delayed Transactions described in the following sections are designed to detect and report both constant and transient data parity errors, and to prevent transient data parity errors from causing a deadlock condition.

If a target detects a data parity error on a write transaction that would otherwise have been handled as a Delayed Transaction, the target is required to do the following:

1. Complete the data phase in which the error occurred by asserting **TRDY#**. If the master is attempting a burst, the target must also assert **STOP#**.
2. Report the error as described in Section 3.7.4.1.
3. Discard the transaction. No Delayed Write Request is enqueued, and no Delayed Write Completion is retired.

If the target detects a data parity error during the initial request phase of a Delayed Write Transaction, no Delayed Request is ever enqueued.

If the target enqueues a good Delayed Write Request and later detects a data parity error during a subsequent repetition of the transaction, the target does not retire any Delayed Write Completions, even if the transaction appears to match one previously enqueued. (It is impossible to determine whether the transaction really matches a previously enqueued one, since an error is present.) This causes the target to have an orphan Delayed Write Completion, because the master believes the transaction has completed, but the target is waiting for the original (error free) request to be repeated. The orphan completion is discarded when the target's Discard Timer expires (refer to Section 3.3.3.3.3.). While waiting for the discard timer to expire, some target implementations will not be able to accept a new Delayed Transaction, since the target is not required to handle multiple Delayed Transactions at the same time. However, since this condition is temporary, a deadlock cannot occur. While in this condition, the device is required to complete transactions that use memory write[25] commands (refer to Section 3.3.3.3.4.).

## 3.7.6. Error Recovery

The action that a system takes as a result of the assertion of **SERR#** is not controlled by this specification. The assertion of **SERR#** by a device indicates that the device has encountered an error from which it cannot recover. The system may optionally stop

[25] This includes two commands: Memory Write and Invalidate and Memory Write.

execution at that point, if it does not have enough information to contain and recover from the error condition.

The PCI parity error signals and status bits are designed to provide a method for data parity errors to be detected and reported (if enabled). On a write transaction, the target always signals data parity errors back to the master on **PERR#**. On a read transaction, the master asserts **PERR#** to indicate to the system that an error was detected. In both cases, the master has the ability to promote the error to its device driver or the operating system or to attempt recovery using hardware and/or software methods.

The system designer may elect to report all data parity errors to the operating system by asserting **SERR#** when the central resource samples **PERR#** asserted. Note that when this option is used, recovery is not possible.

### Implementation Note: Recovery from Data Parity Errors

It is optional for PCI masters and systems to attempt recovery from data parity errors. The following are examples of how data parity error recovery may be attempted:

- **Recovery by the master.** If the master of the transaction in which the parity error was detected has sufficient knowledge that the transaction can be repeated without side-effects, then the master may simply repeat the transaction. If no error occurs on the repeated transaction, reporting of the parity error (to the operating system or device driver) is unnecessary. If the error persists, or if the master is not capable of recovering from the data parity error, the master must inform its device driver. This can be accomplished by generating an interrupt, modifying a status register, setting a flag, or other suitable means. When the master does not have a device driver, it may report the error by asserting **SERR#**.

  Note: Most devices have side-effects when accessed, and, therefore, it is unlikely that recovery is possible by simply repeating a transaction. However, in applications where the master understands the behavior of the target, it may be possible to recover from the error by repetition of the transaction.

- **Recovery by the device driver.** The device driver may support an error recovery mechanism such that the data parity error can be corrected. In this case, the reporting of the error to the operating system is not required. For example, the driver may be able to repeat an entire block transfer by reloading the master with the transfer size, source, and destination addresses of the data. If no error occurs on the repeated block transfer, then the error is not reported. When the device driver does not have sufficient knowledge that the access can be repeated without side-effects, it must report the error to the operating system.

- **Recovery (or error handling) by the operating system.** Once the data parity error has been reported to the operating system, no other agent or mechanism can recover from the error. How the operating system handles the data parity error is operating system dependent.

## 3.8. 64-Bit Bus Extension

PCI supports a high 32-bit bus, referred to as the 64-bit extension to the standard low 32-bit bus. The 64-bit bus provides additional data bandwidth for agents that require it. The high 32-bit extension for 64-bit devices needs an additional 39 signal pins: **REQ64#**, **ACK64#**, **AD[63::32]**, **C/BE[7::4]#**, and **PAR64**. These signals are defined in Section 2.2.9. 32-bit agents work unmodified with 64-bit agents. 64-bit agents must

default to 32-bit mode unless a 64-bit transaction is negotiated. Hence, 64-bit transactions are totally transparent to 32-bit devices. Note: 64-bit addressing does not require a 64-bit data path (refer to Section 3.9.).

64-bit transactions on PCI are dynamically negotiated (once per transaction) between the master and target. This is accomplished by the master asserting **REQ64#** and the target responding to the asserted **REQ64#** by asserting **ACK64#**. Once a 64-bit transaction is negotiated, it holds until the end of the transaction. **ACK64#** must not be asserted unless **REQ64#** was sampled asserted during the same transaction. **REQ64#** and **ACK64#** are externally pulled up to ensure proper behavior when mixing 32- and 64-bit agents. Refer to Section 3.8.1. for the operation of 64-bit devices in a 32-bit system.

During a 64-bit transaction, all PCI protocol and timing remain intact. Only memory transactions make sense when doing 64-bit data transfers. Interrupt Acknowledge and Special Cycle[26] commands are basically 32-bit transactions and must not be used with a **REQ64#**. The bandwidth requirements for I/O and configuration transactions cannot justify the added complexity, and, therefore, only memory transactions support 64-bit data transfers.

All memory transactions and other bus transfers operate the same whether data is transferred 32 or 64 bits at a time. 64-bit agents can transfer from one to eight bytes per data phase, and all combinations of byte enables are legal. As in 32-bit mode, byte enables may change on every data phase. The master initiating a 64-bit data transaction must use a double DWORD (Quadword or 8 byte) referenced address (**AD[2]** must be "0" during the address phase).

When a master requests a 64-bit data transfer (**REQ64#** asserted), the target has three basic responses and each is discussed in the following paragraphs.

1. Complete the transaction using the 64-bit data path (**ACK64#** asserted).
2. Complete the transaction using the 32-bit data path (**ACK64#** deasserted).
3. Complete a single 32-bit data transfer (**ACK64#** deasserted, **STOP#** asserted).

The first option is where the target responds to the master that it can complete the transaction using the 64-bit data path by asserting **ACK64#**. The transaction then transfers data using the entire data bus and up to 8 bytes can be transferred in each data phase. It behaves like a 32-bit bus except more data transfers each data phase.

The second option occurs when the target cannot perform a 64-bit data transfer to the addressed location (it may be capable in a different space). In this case, the master is required to complete the transaction acting as a 32-bit master and not as a 64-bit master. The master has two options when the target does not respond by asserting **ACK64#** when the master asserts **REQ64#** to start a write transaction. The first option is that the master quits driving the upper **AD** lines and only provides data on the lower 32 **AD** lines. The second option is the master continues presenting the full 64 bits of data on each even DWORD address boundary. On the odd DWORD address boundary, the master drives the same data on both the upper and lower portions of the bus.

The third and last option is where the target is only 32 bits and cannot sustain a burst for this transaction. In this case, the target does not respond by asserting **ACK64#**, but terminates the transaction by asserting **STOP#**. If this is a Retry termination (**STOP#** asserted and **TRDY#** deasserted) the master repeats the same request (as a 64-bit request)

[26] Since no agent claims the access by asserting **DEVSEL#** and, therefore, cannot respond with **ACK64#**.

at a later time. If this is a Disconnect termination (**STOP#** and **TRDY#** asserted), the master must repeat the request as a 32-bit master since the starting address is now on a odd DWORD boundary. If the target completed the data transfer such that the next starting address would be a even DWORD boundary, the master would be free to request a 64-bit data transfer. Caution should be used when a 64-bit request is presented and the target transfers a single DWORD as a 32-bit agent. If the master were to continue the burst with the same address, but with the lower byte enables deasserted, no forward progress would be made because the target would not transfer any new data, since the lower byte enables are deasserted. Therefore, the transaction would continue to be repeated forever without making progress.

64-bit parity (**PAR64**) works the same for the high 32-bits of the 64-bit bus as the 32-bit parity (**PAR**) works for the low 32-bit bus. **PAR64** covers **AD[63::32]** and **C/BE[7::4]#** and has the same timing and function as **PAR**. (The number of "1"s on **AD[63::32]**, **C/BE[7::4]#**, and **PAR64** equal an even number). **PAR64** must be valid one clock after each address phase on any transaction in which **REQ64#** is asserted. (All 64-bit targets qualify address parity checking of **PAR64** with **REQ64#**.) 32-bit devices are not aware of activity on 64-bit bus extension signals.

For 64-bit devices checking parity on data phases, **PAR64** must be additionally qualified with the successful negotiation of a 64-bit transaction. **PAR64** is required for 64-bit data phases; it is not optional for a 64-bit agent.

In the following two figures, a 64-bit master requests a 64-bit transaction utilizing a single address phase. This is the same type of addressing performed by a 32-bit master (in the low 4 GB address space). The first, Figure 3-21, is a read where the target responds with **ACK64#** asserted and the data is transferred in 64-bit data phases. The second, Figure 3-22, is a write where the target does not respond with **ACK64#** asserted and the data is transferred in 32-bit data phases (the transaction defaulted to 32-bit mode). These two figures are identical to Figures 3-5 and 3-6 except that 64-bit signals have been added and in Figure 3-21 data is transferred 64-bits per data phase. The same transactions are used to illustrate that the same protocol works for both 32- and 64-bit transactions.

**AD[63::32]** and **C/BE[7::4]#** are reserved during the address phase of a single address phase transaction. **AD[63::32]** contain data and **C/BE[7::4]#** contain byte enables for the upper four bytes during 64-bit data phases of these transactions. **AD[63::32]** and **C/BE[7::4]#** are defined during the two address phases of a dual address cycle (DAC) and during the 64-bit data phases (refer to Section 3.9. for details).

Figure 3-21 illustrates a master requesting a 64-bit read transaction by asserting **REQ64#** (which exactly mirrors **FRAME#**). The target acknowledges the request by asserting **ACK64#** (which mirrors **DEVSEL#**). Data phases are stretched by both agents deasserting their ready lines. 64-bit signals require the same turnaround cycles as their 32-bit counterparts.

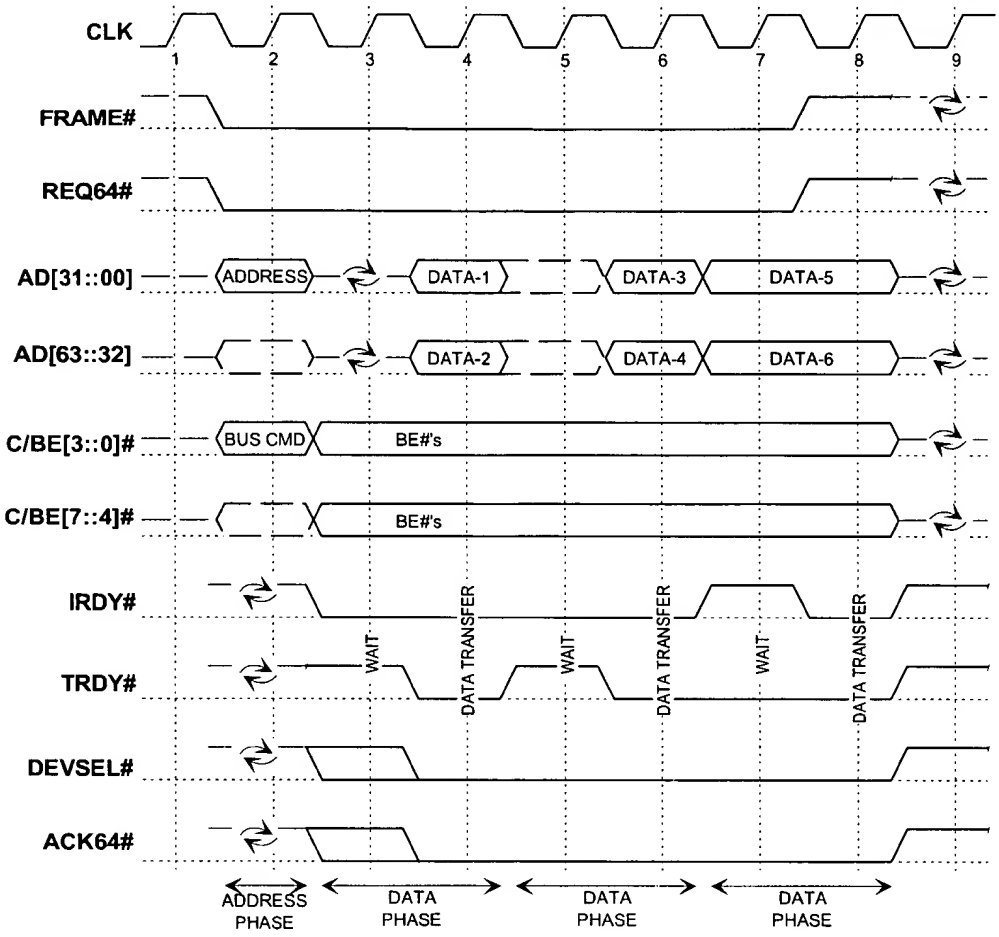



**Figure 3-21: 64-bit Read Request With 64-bit Transfer**

Figure 3-22 illustrates a master requesting a 64-bit transfer. The 32-bit target is not connected to **REQ64#** or **ACK64#**, and **ACK64#** is kept in the deasserted state with a pull-up. As far as the target is concerned, this is a 32-bit transfer. The master converts the transaction from 64- to 32-bits. Since the master is converting 64-bit data transfers into 32-bit data transfers, there may or may not be any byte enables asserted during any data phase of the transaction. Therefore, all 32-bit targets must be able to handle data phases with no byte enables asserted. The target should not use Disconnect or Retry because a data phase is encountered that has no asserted byte enables, but should assert **TRDY#** and complete the data phase. However, the target is allowed to use Retry or Disconnect because it is internally busy and unable to complete the data transfer independent of which byte enables are asserted. The master resends the data that originally appeared on **AD[63::32]** during the first data phase on **AD[31::00]** during the second data phase. The subsequent data phases appear exactly like the 32-bit transfer. (If the 64-bit signals are removed, Figure 3-22 and Figure 3-6 are identical.)

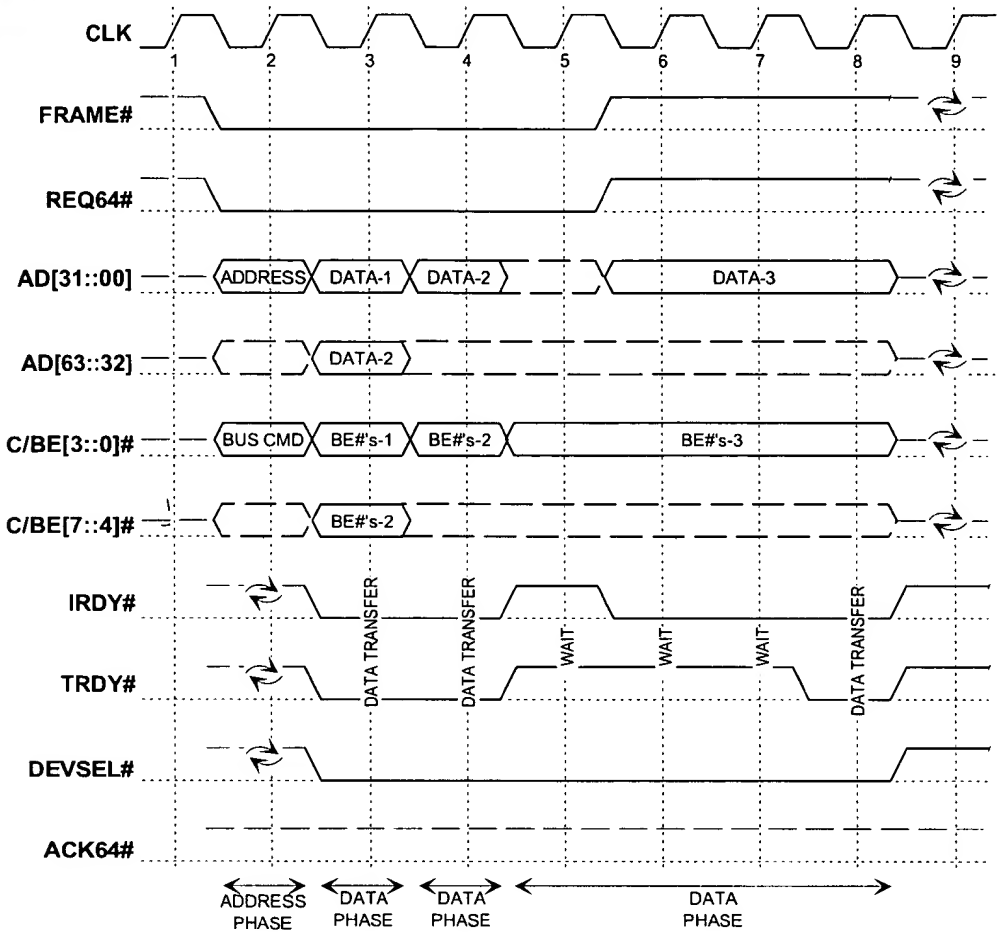



**Figure 3-22: 64-bit Write Request With 32-bit Transfer**

Using a single data phase with 64-bit transfers may not be very effective. Since the master does not know how the transaction will be resolved with **ACK64#** until **DEVSEL#** is returned, it does not know the clock on which to deassert **FRAME#** for a 64-bit single data phase transaction. **IRDY#** must remain deasserted until **FRAME#** signaling is resolved. The single 64-bit data phase may have to be split into two 32-bit data phases when the target is only 32-bits, which means a two phase 32-bit transfer is at least as fast as a one phase 64-bit transfer.

## 3.8.1. Determining Bus Width During System Initialization

**REQ64#** is used during reset to distinguish between parts that are connected to a 64-bit data path, and those that are not. PCI expansion slots that support only a 32-bit data path must not connect **REQ64#** to any other slots or devices. (The **REQ64#** and **ACK64#** pins are located in the 32-bit portion of the connector.) Each 32-bit-only connector must have an individual pull-up resistor for **REQ64#** on the motherboard. **ACK64#** is bused to all 64-bit devices and slots on the motherboard and pulled up with a single resistor located on the motherboard. **ACK64#** for each 32-bit slots must be deasserted either by connecting it to the **ACK64#** signal connecting the 64-bit devices and slots or by individual pull-up resistors on the motherboard.

**REQ64#** is bused to all devices on the motherboard (including PCI connector slots) that support a 64-bit data path. This signal has a single pull-up resistor on the motherboard. The central resource must drive **REQ64#** low (asserted) during the time that **RST#** is asserted, according to the timing specification in Section 4.3.2. Devices that see **REQ64#** asserted on the rising edge of **RST#** are connected to the 64-bit data path, and those that do not see **REQ64#** asserted are not connected. This information may be used by the component to stabilize floating inputs during runtime, as described below.

**REQ64#** has setup and hold time requirements relative to the deasserting (high-going) edge of **RST#**. While **RST#** is asserted, **REQ64#** is asynchronous with respect to **CLK**.

When a 64-bit data path is provided, **AD[63::32]**, **C/BE[7::4]#**, and **PAR64** require either pull-up resistors or input "keepers," because they are not used in transactions with 32-bit devices and may, therefore, float to the threshold level causing oscillation or high power drain through the input buffer. This pull-up or keeper function must be part of the motherboard central resource, not the expansion board, (refer to Section 4.3.3.) to ensure a consistent solution and avoid pull-up current overload.

When the 64-bit data path is present on a device but not connected (as in a 64-bit card plugged into a 32-bit PCI slot), that PCI device must insure that its inputs do not oscillate, and that there is not a significant power drain through the input buffer both before and after the rising edge of **RST#**. This can be done in a variety of ways; e.g., biasing the input buffer or actively driving the outputs continuously (since they are not connected to anything). External resistors on an expansion board or any solution that violates the input leakage specification are prohibited.

While **RST#** is asserted, the PCI device floats its output buffers for the extended data path, **AD[63::32]**, **C/BE[7::4]#**, and **PAR64**, unless the device input buffers cannot tolerate their inputs floating for an indefinitely long **RST#** period. If the device input buffers cannot tolerate this, the component must control its inputs while **RST#** is asserted. In this case, the device is permitted to enable its outputs continuously while **RST#** is asserted and **REQ64#** is deasserted (indicating a 32-bit bus), but must drive them to a logic low level (in case the bus connection is actually 64-bits wide and **REQ64#** has not yet settled to its final value). After the device detects that **REQ64#** is deasserted at the rising edge of **RST#**, the device must continue to control the extended bus to protect the device input buffers.

## 3.9. 64-bit Addressing

PCI supports memory addressing beyond the low 4 GB by defining a mechanism to transfer a 64-bit address from the master of the transaction to the target. No additional pins are required for a 32- or 64-bit device to support 64-bit addressing. Devices that support only 32-bit addresses are mapped into the low 4 GB of the address space and work transparently with devices that generate 64-bit addresses. Only memory transactions support 64-bit addressing.

The width of the address is independent of the width of the bus on either the master or the target. If both the master and target support a 64-bit bus, the entire 64-bit address could theoretically be provided in a single clock. However, the master is required in all cases to use two clocks to communicate a 64-bit address, since the width of the target's bus is not known during the address phase.

The standard PCI bus transaction supports a 32-bit address, Single Address Cycle (SAC), where the address is valid for a single clock when **FRAME#** is first sampled asserted.

To support the transfer of a 64-bit address, a Dual Address Cycle (DAC) bus command is used, accompanied with one of the defined bus commands to indicate the desired data phase activity for the transaction. The DAC uses two clocks to transfer the entire 64-bit address on the **AD[31::00]** signals. Masters that use address stepping cannot implement 64-bit addressing since there is no mechanism for delaying or extending the second address phase. When a 64-bit master uses DAC (64-bit addressing), it must provide the upper 32 bits of the address on **AD[63::32]** and the associated command for the transaction on **C/BE[7::4]#** during both address phases of the transaction to allow 64-bit targets additional time to decode the transaction.

Figure 3-23 illustrates a DAC for a read transaction. In a basic SAC read transaction, a turnaround cycle follows the address phase. In the DAC read transaction, an additional address phase is inserted between the standard address phase and the turnaround cycle. In the figure, the first and second address phases occur on clock 2 and 3 respectively. The turnaround cycle between the address and data phases is delayed until clock 4. Note: **FRAME#** must be asserted during both address phases even for nonbursting single data phase transactions. To adhere to the **FRAME# - IRDY#** relationship, **FRAME#** cannot be deasserted until **IRDY#** is asserted. **IRDY#** cannot be asserted until the master provides data on a write transaction or is ready to accept data on a read transaction.

A DAC is decoded by a potential target when a "1101" is present on **C/BE[3::0]#** during the first address phase. If a 32-bit target supports 64-bit addressing, it stores the address that was transferred on **AD[31::00]** and prepares to latch the rest of the address on the next clock. The actual command used for the transaction is transferred during the second address phase on **C/BE[3::0]#**. A 64-bit target is permitted to latch the entire address on the first address phase. Once the entire address is transferred and the command is latched, the target determines if **DEVSEL#** is to be asserted. The target can do fast, medium, or slow decode one clock delayed from SAC decoding. A subtractive decode agent adjusts to the delayed device selection timing either by ignoring the entire transaction or by delaying its own assertion of **DEVSEL#**. If the bridge does support 64-bit addressing, it will delay asserting its **DEVSEL#** (if it does support 64-bit addressing). The master (of a DAC) will also delay terminating the transaction with Master-Abort for one additional clock.

The execution of an exclusive access is the same for either DAC or SAC. In either case, **LOCK#** is deasserted during the address phase (first clock) and asserted during the second clock (which is the first data phase for SAC and the second address phase for a DAC). Agents monitoring the transaction understand the lock resource is busy, and the target knows the master is requesting a locked operation. For a target that supports both SAC and DAC, the logic that handles **LOCK#** is the same.

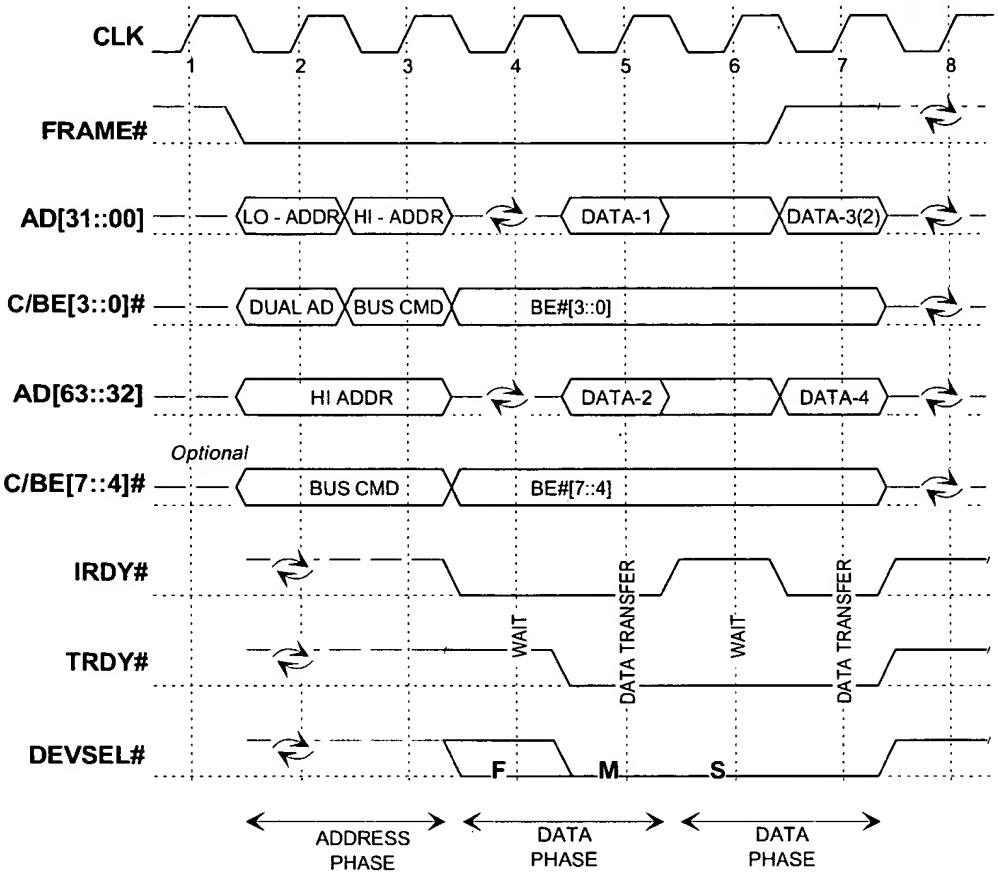



**Figure 3-23. 64-Bit Dual Address Read Cycle**

The master communicates a 64-bit address as shown in Figure 3-23, regardless of whether the target supports a 32-bit or 64-bit bus. The shaded area in Figure 3-23 is used only when the master of the access supports a 64-bit bus. The master drives the entire address (lower address on **AD[31::00]** and upper address on **AD[63::32]**) and both commands (DAC "1101" on **C/BE[3::0]#** and the actual bus command on **C/BE[7::4]#**), all during the initial address phase. On the second address phase, the master drives the upper address on **AD[31::00]** (and **AD[63::32]**) while the bus command is driven on **C/BE[3::0]#** (and **C/BE[7::4]#**). The master cannot determine if the target supports a 64-bit data path until the entire address has been transferred and, therefore, must assume a 32-bit target while providing the address.

If both the master and target support a 64-bit bus, then 64-bit addressing causes no additional latency when determining **DEVSEL#**, since all required information for command decoding is supplied in the first address phase. For example, a 64-bit target that normally performs a medium **DEVSEL#** decode for a SAC can decode the full 64-bit address from a 64-bit master during the first address phase of the DAC and perform a fast **DEVSEL#** decode. If either the master or the target does not support a 64-bit data path, one additional clock of delay will be encountered.

A master that supports 64-bit addressing must generate a SAC, instead of a DAC, when the upper 32 bits of the address are zero. This allows masters that generate 64-bit addresses to communicate with 32-bit addressable targets via SAC. The type of addressing (SAC or DAC) depends on whether the address is in the low 4-GB address range or not, and not by the target's bus width capabilities.

A 64-bit addressable target must act like a 32-bit addressable target (respond to SAC transactions) when mapped in the lower 4 GB address space. If a 32-bit master must access targets mapped above the lower 4 GB address space, that master must support 64-bit addressing using DAC.

## 3.10. Special Design Considerations

This section describes topics that merit additional comments or are related to PCI but are not part of the basic operation of the bus.

1. **Third party DMA**
   Third party DMA is not supported on PCI since sideband signals are not supported on the connector. The intent of PCI is to group together the DMA function in devices that need master capability and, therefore, third party DMA is not supported.

2. **Snooping PCI transactions**
   Any transaction generated by an agent on PCI may be snooped by any other agent on the same bus segment. Snooping does not work when the agents are on different PCI bus segments. In general, the snooping agent cannot drive any PCI signal, but must be able to operate independently of the behavior of the current master or target.

3. **Illegal protocol behavior**
   A device is not encouraged actively to check for protocol errors. However, if a device does detect illegal protocol events (as a consequence of the way it is designed), the design may return its state machines (target or master) to an Idle state as quickly as possible in accordance with the protocol rules for deassertion and tri-state of signals driven by the device.

4. **VGA palette snoop**
   The active VGA device always responds to a read of the color palette, while either the VGA or graphics agent will be programmed to respond to write transactions to the color palette and the other will snoop it. When a device (VGA or graphics) has been programmed to snoop a write to the VGA palette register, it must only latch the data when **IRDY#** and **TRDY#** are both asserted on the same rising clock edge or when a Master-Abort occurs. The first option is the normal case when a VGA and graphics device are present in the same system. The second option occurs when no device on the current bus has been programmed to positively respond to this range of addresses. This occurs when the PCI segment is given the first right of refusal and a subtractive decode device is not present. In some systems, this access is still forwarded to another bus which will complete the access. In this type of system, a device that has been programmed to snoop writes to the palette should latch the data when the transaction is terminated with Master-Abort.

   The palette snoop bit will be set by the system BIOS when it detects both a VGA device and a graphics accelerator device that are on separate boards on the same bus or on the same path but on different buses.

   - When both agents are PCI devices that reside on the same bus, either device can be set to snoop and the other will be set to positively respond.
   - When both are PCI devices that reside on different buses but on the same path, the first device found in the path will be set to snoop and the other device may be set to positively respond or snoop the access. (Either option works in a PC-AT compatible system since a write transaction on a PCI segment, other than the primary PCI bus, that is terminated with Master-Abort is simply terminated and the data is dropped and Master-Aborts are not reported.)

- When one device is on PCI and the other is behind the subtractive decode device, such as an ISA, EISA, or Micro Channel bridge, the PCI device will be set to snoop and the subtractive decode device will automatically claim the access and forward it.

The only case where palette snooping would be turned off is when only a VGA device (no graphics device) is present in the system, or both the VGA and graphics devices are integrated together into single device or card.

Note: Palette snooping does not work when the VGA and graphics devices reside on different buses that are not on the same path. This occurs because only a single agent per bus segment may claim the access. Therefore, one agent will never see the access because its bridge cannot forward the access. When a device has been programmed to snoop the access, it cannot insert wait states or delay the access in any way and, therefore, must be able to latch and process the data without delay.

For more information on PCI support of VGA devices, refer to Appendix A of the *PCI-to-PCI Bridge Architecture Specification.*

5. **Potential deadlock scenario when using PCI-to-PCI bridges**
Warning: A potential deadlock will occur when all the following conditions exist in a system:

   1. When PCI-to-PCI bridges are supported in the system. (Note: If a PCI expansion board connector is supported, PCI-to-PCI bridges may be present in the system.)
   2. A read access originated by the host bridge targets a PCI device that requires more than a single data phase to complete. (Eight-byte transfer or an access that crosses a DWORD boundary when targeting an agent that responds to this request as 32-bit agent or resides on a 32-bit PCI segment.)

   The deadlock occurs when the following steps are met:

   1. A burst read is initiated on PCI by the host bridge and only the first data phase completes. (This occurs because either the target or the PCI-to-PCI bridge in the path terminates the request with Disconnect.)
   2. The request passes through a PCI-to-PCI bridge and the PCI-to-PCI bridge allows posted write data (moving toward main memory) after the initial read completes.
   3. The host bridge that originated the read request blocks the path to main memory.

   The deadlock occurs because the PCI-to-PCI bridge cannot allow a read to transverse it while holding posted write data. The host bridge that initiated the PCI access cannot allow the PCI-to-PCI bridge to flush data until it completes the second read, because there is no way to "back-off" the originating agent without losing data. It must be assumed the read data was obtained from a device that has destructive read side-effects. Therefore, discarding the data and repeating the access is not an option.

   If all these conditions are met, the deadlock will occur. If the system allows all the conditions to exist, then the host bridge initiating the read request must use **LOCK#** to guarantee that the read access will complete without the deadlock conditions being met. The fact that **LOCK#** is active for the transaction causes the PCI-to-PCI bridge to turn-off posting until the lock operation completes. (A locked operation completes when **LOCK#** is deasserted while **FRAME#** is deasserted.)

Note: The use of **LOCK#** is only supported by PCI-to-PCI bridges moving downstream (away from the processor). Therefore, this solution is only applicable to host bus bridges.

Another deadlock that is similar to the above deadlock occurs doing an I/O Write access that straddles an odd DWORD boundary. The same condition occurs as the read deadlock when the host bridge cannot allow access to memory until the I/O write completes. However, **LOCK#** cannot be used to prevent this deadlock since locked accesses must be initiated with a read access.

6. **Potential data inconsistency when an agent uses delayed transaction termination**

Delayed Completion transactions on PCI are matched by the target with the requester by comparing addresses, bus commands, and byte enables, and if a write, write data. As a result, when two masters access the same address with the same bus command and byte enables, it is possible that one master will obtain the data assuming that it is a read which was actually requested by the other master. In a prefetchable region, this condition can occur even if the byte enables, and in some cases, the commands of the two transactions do not match. A prefetchable region can be defined by the target using range registers or by the master using the Memory Read Line or Memory Read Multiple commands. Targets completing read accesses in a prefetchable memory range ignore the byte enables and can also alias the memory read commands when completing the delayed read request.

If no intervening write occurs between the read issued by the two masters, there is no data consistency issue. However, if a master completes a memory write and then requests a read of the same location, there is a possibility that the read will return a snapshot of that location which actually occurred prior to the write (due to a Delayed Read Request by another master queued prior to the write).

This is only a problem when multiple masters on one side of a bridge are polling the same location on the other side of the bridge and one of the masters also writes the location. Although it is difficult to envision a real application with these characteristics, consider the sequence below:

1. Master A attempts a read to location X and a bridge responds to the request using Delayed Transaction semantics (queues a Delayed Read Request).
2. The bridge obtains the requested read data and the Delayed Request is now stored as a Delayed Completion in the bridge.
3. Before Master A is able to complete the read request (obtain the results stored in the Delayed Completion in the bridge), Master B does a memory write to Location X and the bridge posts the memory write transaction.
4. Master B then reads location X using the same address, byte enables, and bus command as Master A's original request. Note that if the transaction reads from a prefetchable location, the two commands can be confused by the bridge even if the byte enable patterns and read commands are different.
5. The bridge completes Master B's read access and delivers read data which is a snapshot of Location X prior to the memory write of Location X by Master B.

Since both transactions are identical, the bridge provides the data to the wrong master. If Master B takes action on the read data, then an error may occur, since Master B will see the value before the write. However, if the purpose of the read by Master B was to ensure that the write had completed at the destination, no error

occurs and the system is coherent since the read data is not used (dummy read). If the purpose of the read is only to flush the write posted data, it is recommended that the read be to a different DWORD location of the same device. Then the reading of stale data does not exist. If the read is to be compared to decide what to do, it is recommended that the first read be discarded and the decision be based on the second read.

The above example applies equally to an I/O controller that uses Delayed Transaction termination. In the above example, replace the word "bridge" with "I/O controller" and the same potential problem exists.

A similar problem can occur if the two masters are not sharing the same location, but locations close to each other, and one master begins reading at a *smaller* address than the one actually needed. If the smaller address coincides exactly with the address of the other master's read from the near location, then the two masters' reads can be swapped by a device using Delayed Transaction termination. If there is an intervening write cycle, then the second master may receive stale data; i.e., the results from the read which occurred before the write cycle. The result of this example is the same as the first example since the start addresses are the same. To avoid this problem, the master must address the data actually required and not start at a smaller address.

In summary, this problem can only occur if two masters on one side of a bridge are sharing locations on the other side of the bridge. Although typical applications are not configured this way, the problem can be avoided if a master doing a read fetches only the actual data it needs and does *not* prefetch data *before* the desired data, or if the master does a dummy read after the write to guarantee that the write completes.

Another data inconsistency situation can occur when a single master changes its behavior based on a new transaction it receives after having a request terminated with Retry. The following sequence illustrates the data inconsistency:

1. A master is informed that pointer 1 at DWORD Location X is valid. (Pointer 2 at Location Y, the next sequential DWORD location, is not valid.)
2. The master initiates a memory read to Location X and is terminated with Retry. (The master intends to read only pointer 1, since pointer 2 is invalid.)
3. The host bridge begins to fetch the contents of Location X as a Delayed Transaction.
4. The host bridge completes the read request, prefetching beyond Location X to include Location Y and places the Delayed Read Completion in the outbound queue.
5. The CPU updates pointer 2 in Location Y in memory.
6. The CPU uses a memory write to inform the master that pointer 2 is valid. The host bridge posts the memory write. Ordering rule number 7 in Appendix E requires the host bridge to allow the posted memory write transaction to pass the Delayed Read Completion of Location X (including the stale value from Location Y).
7. The host bridge executes the posted memory write on the PCI bus informing the master that pointer 2 in now valid.
8. The master repeats the original memory read to Location X, but because pointer 2 in now valid, it extends the transaction and obtains two DWORDS including Location Y.

The data the master received from Location Y is stale. To prevent this data inconsistency from occurring, the master is not allowed to extend a memory read transaction beyond its original intended limits after it has been terminated with Retry.

7. **Peer-to-peer transactions crossing multiple host bridges**

   PCI host bridges may, but are not required to, support PCI peer-to-peer transactions that traverse multiple PCI host bridges.

8. **The effect of PCI-to-PCI bridges on the PCI clock specification**

   The timing parameters for **CLK** for PCI expansion connectors are specified at the input of the device in the slot. Refer to Section 4.2.3.1. and Section 7.6.4.1. for more information. Like all signals on the connector, only a single load is permitted on **CLK** in each slot. An expansion board that uses several devices behind a PCI-to-PCI bridge must accommodate the clock buffering requirements of that bridge. For example, if the bridge's clock buffer affects the duty cycle of **CLK**, the rest of the devices on the expansion board must accept the different duty cycle. It is the responsibility of the expansion board designer to choose components with compatible **CLK** specifications.

   The system must always guarantee the timing parameters for **CLK** specified in Section 4.2.3.1. and Section 7.6.4.1. at the input of the device in a PCI expansion slot, even if the motherboard places PCI expansion slots on the secondary side of a PCI-to-PCI bridge. It is the responsibility of the motherboard designer to choose clock sources and PCI-to-PCI bridges that will guarantee this specification for all slots.

9. **Devices cannot drive and receive signals at the same time**

   Bus timing requires that no device both drive and receive a signal on the bus at the same time. System timing analysis considers the worst signal propagation case to be when one device drives a signal and the signal settles at the input of all other devices on the bus. In most cases, the signal will not settle at the driving device until some time after it has settled at all other devices. Refer to Section 4.3.5. and Section 7.7.5. for a description of $T_{prop}$.

   Logic internal to a device must never use the signal received from the bus while that device is driving the bus. If internal logic requires the state of a bus signal while the device is driving the bus, that logic must use the internal signal (the one going to the output buffer of the device) rather than the signal received from the device input buffer. For example, if logic internal to a device continuously monitors the state of **FRAME#** on the bus, that logic must use the signal from the device input buffer when the device is not the current bus master, and it must use the internally generated **FRAME#** when the device is the current bus master.


# Appendix E
# System Transaction Ordering

Many programming tasks, especially those controlling intelligent peripheral devices common in PCI systems, require specific events to occur in a specific order. If the events generated by the program do not occur in the hardware in the order intended by the software, a peripheral device may behave in a totally unexpected way. PCI transaction ordering rules are written to give hardware the flexibility to optimize performance by rearranging certain events that do not affect device operation, yet strictly enforce the order of events that do affect device operation.

One performance optimization that PCI systems are allowed to do is the posting of memory write transactions. Posting means the transaction is captured by an intermediate agent; e.g., a bridge from one bus to another, so that the transaction completes at the source before it actually completes at the intended destination. This allows the source to proceed with the next operation while the transaction is still making its way through the system to its ultimate destination.

While posting improves system performance, it complicates event ordering. Since the source of a write transaction proceeds before the write actually reaches its destination, other events that the programmer intended to happen after the write may happen before the write. Many of the PCI ordering rules focus on posting buffers requiring them to be flushed to keep this situation from causing problems.

If the buffer flushing rules are not written carefully, however, deadlock can occur. The rest of the PCI transaction ordering rules prevent the system buses from deadlocking when posting buffers must be flushed.

Simple devices do not post outbound transactions. Therefore, their requirements are much simpler than those presented here for bridges. Refer to Section 3.2.5.1. for the requirements for simple devices.

The focus of the remainder of this appendix is on a PCI-to-PCI bridge. This allows the same terminology to be used to describe a transaction initiated on either interface and is easier to understand. To apply these rules to other bridges, replace a PCI transaction type with its equivalent transaction type of the host bus (or other specific bus). While the discussion focuses on a PCI-to-PCI bridge, the concepts can be applied to all bridges.

The ordering rules for a specific implementation may vary. This appendix covers the rules for all accesses traversing a bridge assuming that the bridge can handle multiple transactions at the same time in each direction. Simpler implementations are possible but are not discussed here.

## E.1 Producer - Consumer Ordering Model

The Producer - Consumer model for data movement between two masters is an example of a system that would require this kind of ordering. In this model, one agent, the Producer, produces or creates the data and another agent, the Consumer, consumes or uses the data. The Producer and Consumer communicate between each other via a flag and a status element. The Producer sets the flag when all the data has been written and then waits for a completion status code. The Consumer waits until it finds the flag set, then it resets the flag, consumes the data, and writes the completion status code. When the Producer finds the completion status code, it clears it and the sequence repeats. Obviously, the order in which the flag and data are written is important. If some of the Producer's data writes were posted, then without buffer-flushing rules it might be possible for the Consumer to see the flag set before the data writes had completed. The PCI ordering rules are written such that no matter which writes are posted, the Consumer can never see the flag set and read the data until the data writes are finished. This specification refers to this condition as "having a consistent view of data." Notice that if the Consumer were to pass information back to the Producer in addition to the status code, the order of writing this additional information and the status code becomes important, just as it was for the data and flag.

In practice, the flag might be a doorbell register in a device or it might be a main-memory pointer to data located somewhere else in memory. And the Consumer might signal the Producer using an interrupt or another doorbell register, rather than having the Producer poll the status element. But in all cases, the basic need remains the same; the Producer's writes to the data area must complete before the Consumer observes that the flag has been set and reads the data.

This model allows the data, the flag, the status element, the Producer, and the Consumer to reside anywhere in the system. Each of these can reside on different buses and the ordering rules maintain a consistent view of the data. For example, in Figure E-1, the agent producing the data, the flag, and the status element reside on Bus 1, while the actual data and the Consumer of the data both reside on Bus 0. The Producer writes the last data and the PCI-to-PCI bridge between Bus 0 and 1 completes the access by posting the data. The Producer of the data then writes the flag changing its status to indicate that the data is now valid for the Consumer to use. In this case, the flag has been set before the final datum has actually been written (to the final destination). PCI ordering rules require that when the Consumer of the data reads the flag (to determine if the data is valid), the read will cause the PCI-to-PCI bridge to flush the posted write data to the final destination before completing the read. When the Consumer determines the data is valid by checking the flag, the data is actually at the final destination.

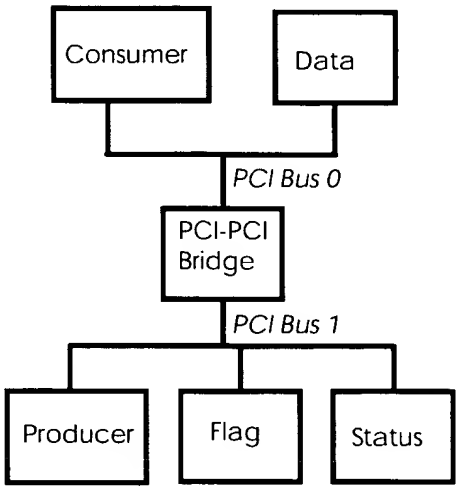



**Figure E-1: Example Producer - Consumer Model**

The ordering rules lead to the same results regardless of where the Producer, the Consumer, the data, the flag, and the status element actually reside. The data is always at the final destination before the Consumer can read the flag. This is true even when all five reside on different bus segments of the system. In one configuration, the data will be forced to the final destination when the Consumer reads the flag. In another configuration, the read of the flag occurs without forcing the data to its final destination; however, the read request of the actual data pushes the final datum to the final destination before completing the read.

A system may have multiple Producer-Consumer pairs operating simultaneously, with different data - flag-status sets located all around the system. But since only one Producer can write to a single data-flag set, there are no ordering requirements between different masters. Writes from one master on one bus may occur in one order on one bus, with respect to another master's writes, and occur in another order on another bus. In this case, the rules allow for some writes to be rearranged; for example, an agent on Bus 1 may see Transaction A from a master on Bus 1 complete first, followed by Transaction B from another master on Bus 0. An agent on Bus 0 may see Transaction B complete first followed by Transaction A. Even though the actual transactions complete in a different order, this causes no problem since the different masters must be addressing different data-flag sets.

## E.2. Summary of PCI Ordering Requirements

Following is a summary of the general PCI ordering requirements presented in Section 3.2.5. These requirements apply to all PCI transactions, whether they are using Delayed Transactions or not.

### General Requirements

1. The order of a transaction is determined when it completes. Transactions terminated with Retry are only requests and can be handled by the system in any order.

2. Memory writes can be posted in both directions in a bridge. I/O and Configuration writes are not posted. (I/O writes can be posted in the Host Bridge, but some restrictions apply.) Read transactions (Memory, I/O, or Configuration) are not posted.
3. Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus.
4. Write transactions crossing a bridge in opposite directions have no ordering relationship.
5. A read transaction must push ahead of it through the bridge any posted writes originating on the *same* side of the bridge and posted *before* the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the *opposite* side and were posted *before* the read command completes on the read-destination bus.
6. A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. Otherwise, a deadlock may occur. Bridges are allowed to refuse to accept a memory write for temporary conditions which are guaranteed to be resolved with time. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of locked transaction as a master only if the bridge has already established a locked operation with its intended target.

The following is a summary of the PCI ordering requirements specific to Delayed Transactions, presented in Section 3.3.3.3.

**Delayed Transaction Requirements**

1. A target that uses Delayed Transactions may be designed to have any number of Delayed Transactions outstanding at one time.
2. Only non-posted transactions can be handled as Delayed Transactions.
3. A master must repeat any transaction terminated with Retry since the target may be using a Delayed Transaction.
4. Once a Delayed Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, it is only a request and may be discarded at any time.
5. A Delayed Completion can only be discarded when it is a read from a prefetchable region, or if the master has not repeated the transaction in $2^{15}$ clocks.
6. A target must accept all memory writes addressed to it, even while completing a request using Delayed Transaction termination.
7. Delayed Requests and Delayed Completions are not required to be kept in their original order with respect to themselves or each other.
8. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.

9. A single master may have any number of outstanding requests terminated with Retry. However, if a master requires one transaction to be completed before another, it cannot attempt the second one on PCI until the first one has completed.

## E.3. Ordering of Requests

A transaction is considered to be a *request* when it is presented on the bus. When the transaction is terminated with Retry, it is still considered a request. A transaction becomes *complete* or a *completion* when data actually transfers (or is terminated with Master-Abort or Target-Abort). The following discussion will refer to transactions as being a request or completion depending on the success of the transaction.

A transaction that is terminated with Retry has no ordering relationship with any other access. Ordering of accesses is only determined when an access completes (transfers data). For example, four masters A, B, C, and D reside on the same bus segment and all desire to generate an access on the bus. For this example, each agent can only request a single transaction at a time and will not request another until the current access completes. The order in which transactions complete are based on the algorithm of the arbiter and the response of the target, not the order in which each agent's **REQ#** signal was asserted. Assuming that some requests are terminated with Retry, the order in which they complete is independent of the order they were first requested. By changing the arbiter's algorithm, the completion of the transactions can be any sequence (i.e., A, B, C, and then D or B, D, C, and then A, and so on). Because the arbiter can change the order in which transactions are requested on the bus, and, therefore, the completion of such transactions, the system is allowed to complete them in any order it desires. This means that a request from any agent has no relationship with a request from any other agent. The only exception to this rule is when **LOCK#** is used, which is described later.

Take the same four masters (A, B, C, and D) used in the previous paragraph and integrate them onto a single piece of silicon (a multi-function device). For a multi-function device, the four masters operate independent of each other, and each function only presents a single request on the bus for this discussion. The order their requests complete is the same as if they where separate agents and not a multi-function device, which is based on the arbitration algorithm. Therefore, multiple requests from a single agent may complete in any order, since they have no relationship to each other.

Another device, not a multi-function device, has multiple internal resources that can generate transactions on the bus. If these different sources have some ordering relationship, then the device must ensure that only a single request is presented on the bus at any one time. The agent must not attempt a subsequent transaction until the previous transaction completes. For example, a device has two transactions to complete on the bus, Transaction A and Transaction B and A must complete before B to preserve internal ordering requirements. In this case, the master cannot attempt B until A has completed.

The following example would produce inconsistent results if it were allowed to occur. Transaction A is to a flag that covers data, and Transaction B accesses the actual data covered by the flag. Transaction A is terminated with Retry, because the addressed target is currently busy or resides behind a bridge. Transaction B is to a target that is ready and will complete the request immediately. Consider what happens when these

two transactions are allowed to complete in the wrong order. If the master allows Transaction B to be presented on the bus after Transaction A was terminated with Retry, Transaction B can complete before Transaction A. In this case, the data may be accessed before it is actually valid. The responsibility to prevent this from occurring rests with the master, which must block Transaction B from being attempted on the bus until Transaction A completes. A master presenting multiple transactions on the bus must ensure that subsequent requests (that have some relationship to a previous request) are not presented on the bus until the previous request has completed. The system is allowed to complete multiple requests from the same agent in any order. When a master allows multiple requests to be presented on the bus without completing, it must repeat each request independent of how any of the other requests complete.

## E.4. Ordering of Delayed Transactions

A Delayed Transaction progresses to completion in three phases:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first phase, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. During the second phase, the target independently completes the request on the destination bus using the latched information from the Delayed Request. The result of completing the Delayed Request on the destination bus produces a Delayed Completion, which consists of the latched information of the Delayed Request and the completion status (and data if a read request). During the third phase, the master successfully re-arbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed.

The number of simultaneous Delayed Transactions a bridge is capable of handling is limited by the implementation and not by the architecture. Table E-1 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. Each column of the table represents an access that was accepted by the bridge earlier, while each row represents a transaction just accepted. The contents of the box indicate what ordering relationship the second transaction must have to the first.

**PMW** - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

**DRR** - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple commands. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted

on the destination bus the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

**DWR** - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write command. Note: Memory Write and Memory Write and Invalidate commands must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Before it is attempted on the destination bus, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

**DRC** - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the status of the target (normal, Master-Abort, Target-Abort, parity error, etc.).

**DWC** - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access, but only status of how it completed (Normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

**No** - indicates that the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes found in column 2 prevent PMW data from being passed by other accesses and thereby maintain a consistent view of data in the system.

**Yes** - indicates that the subsequent transaction must be allowed to complete before the previous one or a deadlock can occur.

When blocking occurs, the PMW is required to pass the Delayed Transaction. If the master continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. There is no ordering violation when these subsequent transactions complete before a prior transaction.

**Yes/No** - indicates that the bridge designer may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

**Table E-1: Ordering Rules for a Bridge**

| Row pass Col.? | PMW (Col 2) | DRR (Col 3) | DWR (Col 4) | DRC (Col 5) | DWC (Col 6) |
|---|---|---|---|---|---|
| **PMW** (Row 1) | No[1] | Yes[5] | Yes[5] | Yes[7] | Yes[7] |
| **DRR** (Row 2) | No[2] | Yes/No | Yes/No | Yes/No | Yes/No |
| **DWR** (Row 3) | No[3] | Yes/No | Yes/No | Yes/No | Yes/No |
| **DRC** (Row 4) | No[4] | Yes[6] | Yes[6] | Yes/No | Yes/No |
| **DWC** (Row 5) | Yes/No | Yes[6] | Yes[6] | Yes/No | Yes/No |

Rule 1 - A subsequent PMW cannot pass a previously accepted PMW. (Col 2, Row 1)

Posted Memory write transactions must complete in the order they are received. If the subsequent write is to the flag that covers the data, the Consumer may use stale data if write transactions are allowed to pass each other.

Rule 2 - A read transaction must push posted write data to maintain ordering. (Col 2, Row 2)

For example, a memory write to a location followed by an immediate memory read of the same location returns the new value (refer to Section 3.10, item 6, for possible exceptions). Therefore, a memory read cannot pass posted write data. An I/O read cannot pass a PMW, because the read may be ensuring the write data arrives at the final destination.

Rule 3 - A non-postable write transaction must push posted write data to maintain ordering. (Col 2, Row 3)

A Delayed Write Request may be the flag that covers the data previously written (PMW), and, therefore, the write flag cannot pass the data that it potentially covers.

Rule 4 - A read transaction must pull write data back to the originating bus of the read transaction. (Col 2, Row 4)

For example, the read of a status register of the device writing data to memory must not complete before the data is pulled back to the originating bus; otherwise, stale data may be used.

Rule 5 - A Posted Memory Write must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Col 3 and Col 4, Row 1)

A deadlock can occur when bridges that support Delayed Transactions are used with bridges that do not support Delayed Transactions. Referring to Figure E-2, a deadlock can occur when Bridge Y (using Delayed Transactions) is between Bridges X and Z (designed to a previous version of this specification and not using Delayed Transactions). Master 1 initiates a read to Target 1 that is forwarded through Bridge X and is queued as a Delayed Request in Bridge Y. Master 3 initiates a read to Target 3 that is forwarded through Bridge Z and is queued as a Delayed Request in Bridge

Y. After Masters 1 and 3 are terminated with Retry, Masters 2 and 4 begin memory write transactions of a long duration addressing Targets 2 and 4 respectively, which are posted in the write buffers of Bridges X and Z respectively. When Bridge Y attempts to complete the read in either direction, Bridges X and Z must flush their posted write buffers before allowing the Read Request to pass through it.

If the posted write buffers of Bridges X and Z are larger than those of Bridge Y, Bridge Y's buffers will fill. If posted write data is not allowed to pass the DRR, the system will deadlock. Bridge Y cannot discard the read request since it has been attempted, and it cannot accept any more write data until the read in the opposite direction is completed. Since this condition exists in both directions, neither DRR can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRR when the DRR blocks forward progress of PMW data.

The same condition exists when a DWR sits at the head of both queues, since some old bridges also require the posting buffers to be flushed on a non-posted write cycle.

Rule 6 – A Delayed Completion (read or write) must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Cols 3 and 4, Rows 4 and 5)

A deadlock can occur when two bridges that support Delayed Transactions are requesting accesses to each other. The common PCI bus segment is on the secondary bus of Bridge A and the primary bus for Bridge B. If neither bridge allows Delayed Completions to pass the Delayed Requests, neither can make progress.

For example, suppose Bridge A's request to Bridge B completes on Bridge B's secondary bus, and Bridge B's request completes on Bridge A's primary bus. Bridge A's completion is now behind Bridge B's request and Bridge B's completion is behind Bridge A's request. If neither bridge allows completions to pass the requests, then a deadlock occurs because neither master can make progress.

Rule 7 - A Posted Memory Write must be allowed to pass a Delayed Completion (read or write) to avoid deadlocks. (Col 5 and Col 6, Row 1)

As in the example for Rule 5, another deadlock can occur in the system configuration in Figure E-2. In this case, however, a DRC sits at the head of the queues in both directions of Bridge Y at the same time. Again the old bridges (X and Z) contain posted write data from another master. The problem in this case, however, is that the read transaction cannot be *repeated* until all the posted write data is flushed out of the old bridge and the master is allowed to repeat its original request. Eventually, the new bridge cannot accept any more posted data because its internal buffers are full and it cannot drain them until the DRC at the other end completes. When this condition exists in both directions, neither DRC can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRC when the DRC blocks forward progress of PMW data.

The same condition exists when a DWC sits at the head of both queues.

**Transactions that have no ordering constraints**

Some transactions enqueued as Delayed Requests or Delayed Completions have no ordering relationship with any other Delayed Requests or Delayed Completions. The designer can (for performance or cost reasons) allow or disallow Delayed Requests to pass other Delayed Requests and Delayed Completions that were previously enqueued.

Delayed Requests can pass other Delayed Requests (Cols 3 and 4, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with other Delayed Requests, these four boxes are don't cares.

Delayed Requests can pass Delayed Completions (Col 5 and 6, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with Delayed Completions, these four boxes are don't cares.

Delayed Completions can pass other Delayed Completions (Col 5 and 6, Rows 4 and 5).

Since Delayed Completions have no ordering relationship with other Delayed Completions, these four boxes are don't cares.

Delayed Write Completions can pass posted memory writes or be blocked by them (Col 2, Row 5).

If the DWC is allowed to pass a PMW or if it remains in the same order, there is no deadlock or data inconsistencies in either case. The DWC data and the PMW data are moving in opposite directions, initiated by masters residing on different buses accessing targets on different buses.

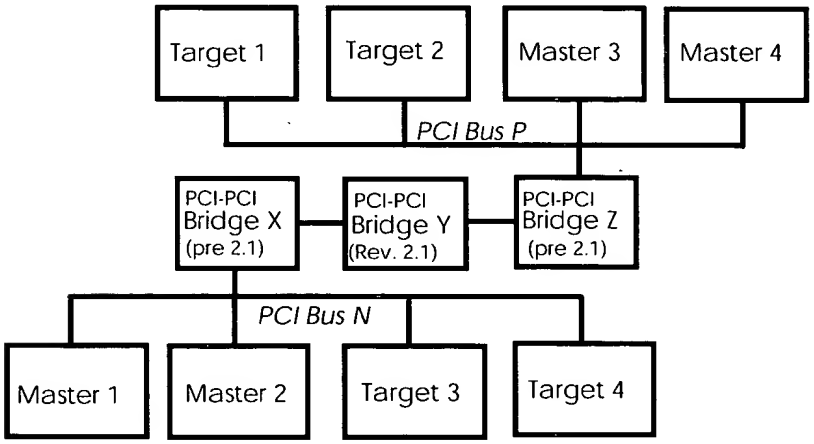



**Figure E-2: Example System with PCI-to-PCI Bridges**

## E.5. Delayed Transactions and LOCK#

The bridge is required to support **LOCK#** when a transaction is initiated on its primary bus (and is using the lock protocol), but is not required to support **LOCK#** on transactions that are initiated on its secondary bus. If a locked transaction is initiated on the primary bus and the bridge is the target, the bridge must adhere to the lock semantics

defined by this specification. The bridge is required to complete (push) all PMWs (accepted from the primary bus) onto the secondary bus before attempting the lock on the secondary bus. The bridge may discard any requests enqueued, allow the locked transaction to pass the enqueued requests, or simply complete all enqueued transactions before attempting the locked transaction on the secondary interface. Once a locked transaction has been enqueued by the bridge, the bridge cannot accept any other transaction from the primary interface until the lock has completed except for a continuation of the lock itself by the lock master. Until the lock is established on the secondary interface, the bridge is allowed to continue enqueuing transactions from the secondary interface, but not the primary interface. Once lock has been established on the secondary interface, the bridge cannot accept any posted write data moving toward the primary interface until **LOCK#** has been released (**FRAME#** and **LOCK#** deasserted on the same rising clock edge). (In the simplest implementation, the bridge does not accept any other transactions in either direction once lock is established on the secondary bus, except for locked transactions from the lock master.) The bridge must complete PMW, DRC, and DWC transactions moving toward the primary bus before allowing the locked access to complete on the originating bus. The preceding rules are sufficient for deadlock free operation. However, an implementation may be more or less restrictive, but, in all cases must ensure deadlock-free operation.

## E.6. Error Conditions

A bridge is free to discard data or status of a transaction that was completed using Delayed Transaction termination when the master has not repeated the request within $2^{10}$ PCI clocks (about 30 µs at 33 MHz). However, it is recommended that the bridge not discard the transaction until $2^{15}$ PCI clocks (about 983 µs at 33 MHz) after it acquired the data or status. The shorter number is useful in a system where a master designed to a previous version of this specification frequently fails to repeat a transaction exactly as first requested. In this case, the bridge may be programmed to discard the abandoned Delayed Completion early and allow other transactions to proceed. Normally, however, the bridge would wait the longer time, in case the repeat of the transaction is being delayed by another bridge or bridges designed to a previous version of this specification that did not support Delayed Transactions.

When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

> Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

When the Discard Timer expires, the device may choose to report or ignore the error. When the data is prefetchable, it is recommended that the device ignore the error since system integrity is not affected. However, when the data is not prefetchable, it is recommended that the device report the error to its device driver since system integrity is affected. A bridge may assert **SERR#** since it typically does not have a device driver.